Compte rendu du TP n°3

Apprentissage non supervisé
Clustering, détection d'anomalie,
réduction de dimensionnalité

Instrumentation

2024-2025

Elaboré par :

Rayane AIT NOURI

Zied MHADHEBI

Introduction :

Dans ce compte rendu, nous présentons les résultats de nos exercices sur l'apprentissage non supervisé, incluant le clustering avec l'algorithme K-means, la détection d'anomalies avec l'algorithme Isolation Forest, et la réduction de dimensionnalité avec l'Analyse en Composantes Principales (ACP). Ces exercices nous ont permis de mettre en pratique les concepts théoriques et de manipuler des jeux de données pour mieux comprendre ces méthodes

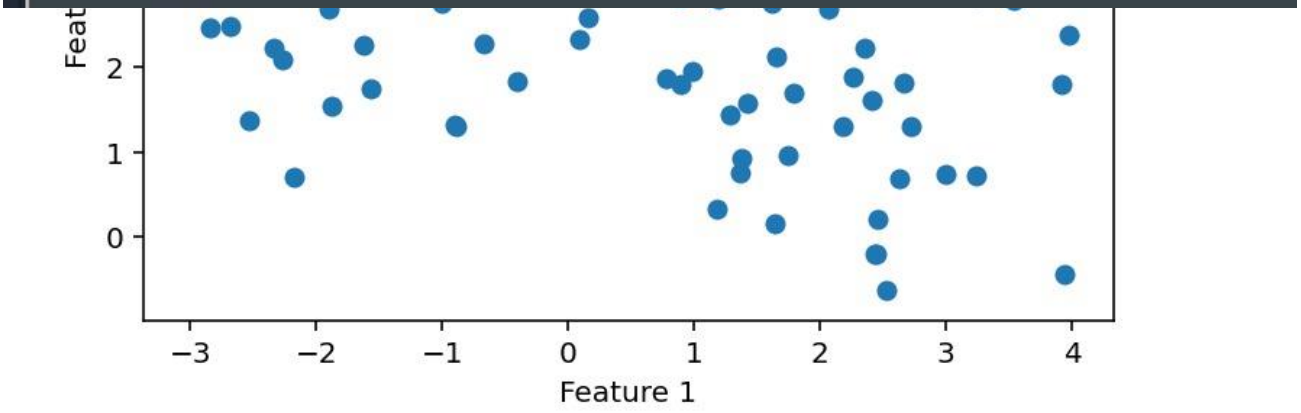
Exercice 1 : Clustering (K-means clustering)

1. Génération et affichage du jeu de données :

Dans cette première étape, nous avons généré un jeu de données composé de 100 points répartis en trois clusters distincts, la visualisation de ces points montre clairement trois groupes bien séparés, ce qui est idéal pour tester l'algorithme K-means.

Code utilisé :

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Génération du jeu de données
X, y = make_blobs(n_samples=100, centers=3, n_features=2, random_state=0)

# Affichage du jeu de données
plt.scatter(X[:, 0], X[:, 1])
plt.title("Jeu de données généré")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```



- Jeu de données généré :

2. Nous avons importé le modèle KMeans de la bibliothèque sklearn.cluster en utilisant la ligne de code suivante :

```
#2)

from sklearn.cluster import KMeans
```

Et nous avons défini le modèle Kmeans avec le paramètre n_clusters=3.

Voici les caractéristiques des hyperparamètres du modèle :

. n_clusters=3 : Ce paramètre signifie que le modèle KMeans va diviser les données en 3 clusters. Cela correspond au nombre de groupes que nous souhaitons identifier dans notre jeu de données.

. Nombre d'initialisations (n_init) : Par défaut, ce nombre est fixé à 10, et cela signifie que l'algorithme KMeans sera exécuté 10 fois avec différentes initialisations des centroïdes, et la meilleure solution parmi ces exécutions sera retenue.

. Nombre d'itérations maximal : Par défaut, le nombre maximal d'itérations est fixé à 300, ce qui signifie que l'algorithme peut s'exécuter jusqu'à 300 itérations pour chaque initialisation, à moins qu'il ne converge avant.

. Méthode d'initialisation : Par défaut, la méthode d'initialisation utilisée est k-means++,, cette méthode choisit les centroïdes initiaux de manière intelligente pour améliorer la convergence et la qualité des clusters.

. Choix des points des centres : Les points des centres sont choisis en utilisant la méthode k-means++, cette méthode sélectionne les centroïdes initiaux de manière à ce qu'ils soient bien répartis dans l'espace des données.

Cela aide à éviter les mauvaises initialisations et à améliorer la convergence de l'algorithme.

3. Entrainement du modèle :

- Ligne de code utilisée :

```
#3)

# Entraînement du modèle
model.fit(X)
```

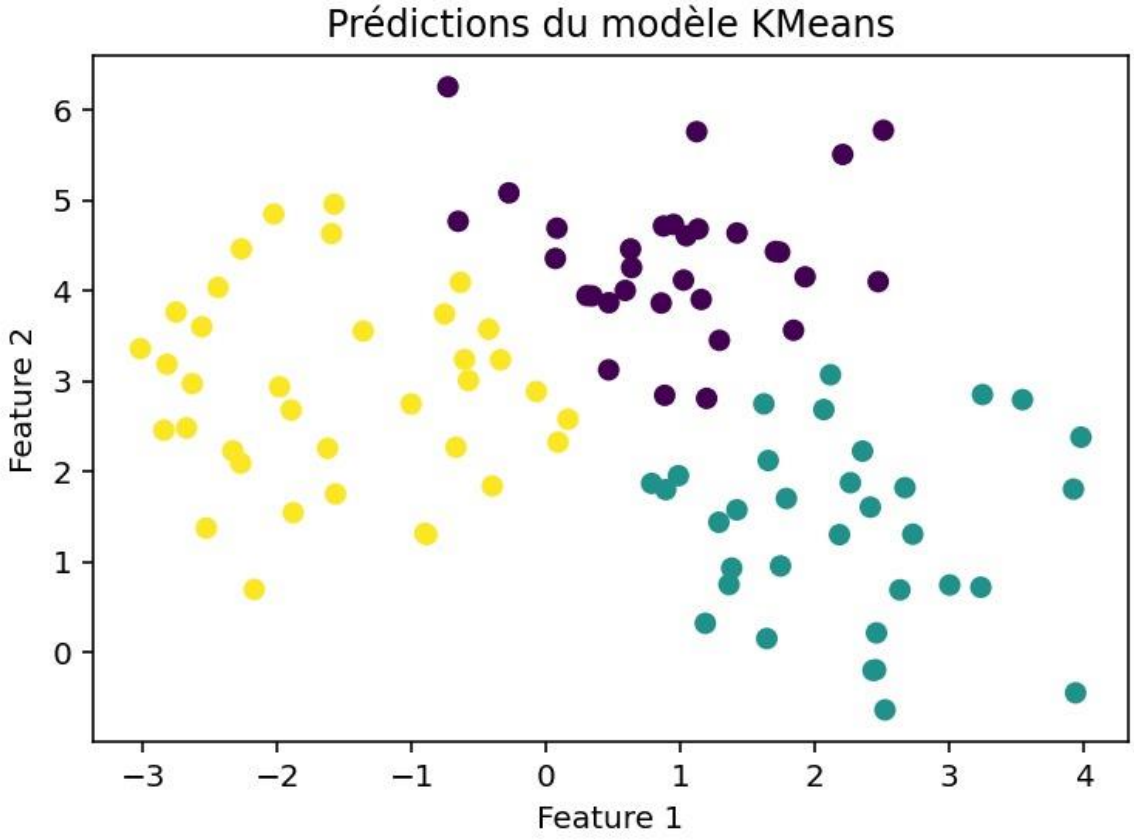
L'entraînement permet de diviser les données en trois clusters distincts, en regroupant les points similaires ensemble.

4. Visualisation de la prédiction du modèle :

- Code utilisé :

```
# Prédiction des clusters
predictions = model.predict(X)

# Affichage des prédictions
plt.scatter(X[:, 0], X[:, 1], c=predictions)
plt.title("Prédictions du modèle KMeans")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

- Résultat :



- Interprétation :

La visualisation des prédictions du modèle KMeans permet de voir comment les points de notre jeu de données ont été assignés aux différents clusters, la méthode model.predict(X) retourne un tableau où chaque élément correspond à l'indice du cluster auquel chaque point a été assigné.

La fonction plt.scatter utilise ces indices pour colorer les points en fonction de leur cluster, chaque couleur représentant un cluster différent. En observant le graphique, nous pouvons voir comment les

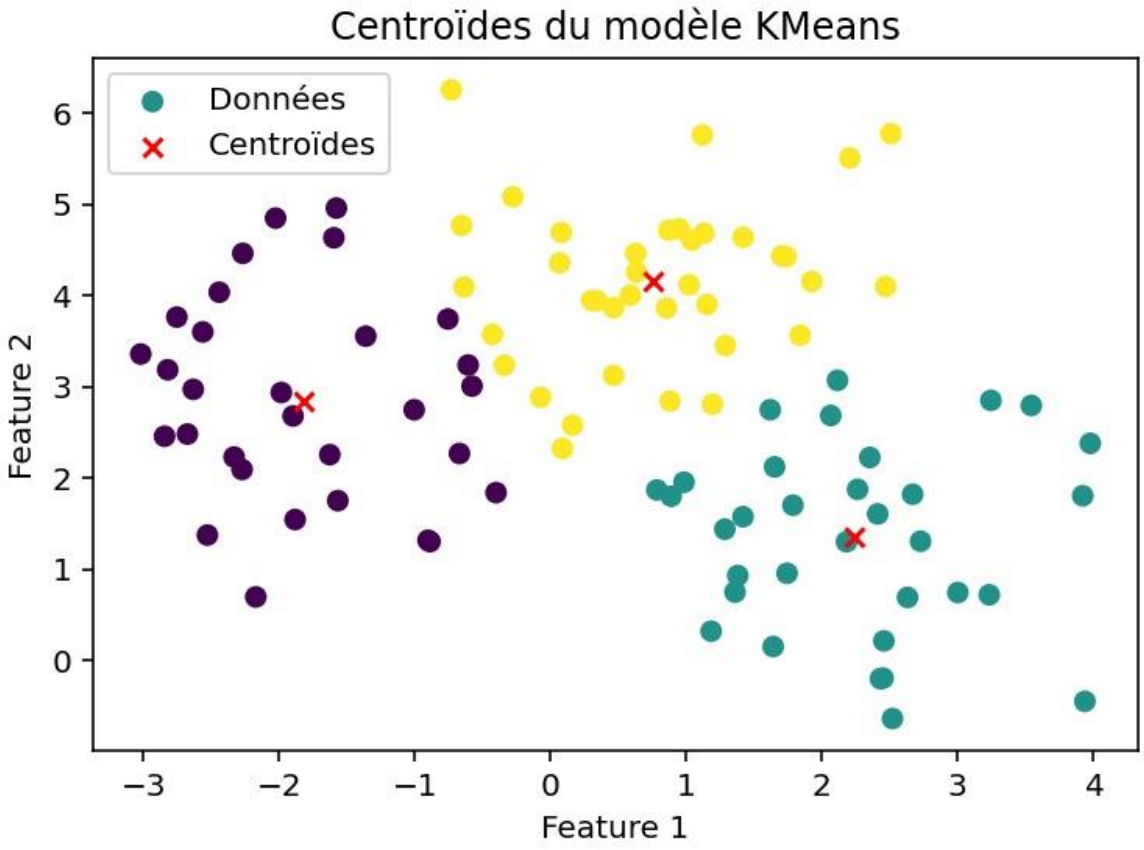
points ont été regroupés : les points de même couleur appartiennent au même cluster, ce qui permet de vérifier visuellement la qualité du clustering.

5. Affichage des centroides :

- Code utilisé :

```
#5)

# Affichage des centroïdes
centroids = model.cluster_centers_
plt.scatter(X[:, 0], X[:, 1], c=predictions, label='Données')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', label='Centroïdes')
plt.title("Centroïdes du modèle KMeans")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

- Résultat :



Les centroïdes sont les centres des clusters formés par l'algorithme KMeans, ils représentent le point moyen de tous les points d'un cluster donné et sont affichés en rouge sur le graphique.

6. Les centrioles sont les points centraux de chaque cluster.

Ils représentent la moyenne des points dans chaque cluster et sont utilisés pour minimiser la variance intra-cluster.

7. Le cout engendré par le modèle :

Le coût (inertia) est calculé comme la somme des distances quadratiques entre chaque point et le centroïde de son cluster.

On peut le calculer en utilisant cette fonction :

$$\sum_{i=0}^{n} \min\left(\|x_i - \mu_j\|^2\right)$$

Avec : x_i : point d'un cluster

μ_j : centre du cluster j

n : nombre de points dans le cluster

En utilisant ce code (code ci-dessou), on a pu trouver le cout qui est égal a 168.62.

```
#7)
# Calcul du coût
inertia = model.inertia_
print("Coût (inertia):", inertia)
```

Cout obtenu :

```
Coût (inertia): 168.62544984385508
```

Le coût du modèle KMeans nous donne une indication de la qualité du clustering. Un coût de 168.62 signifie que les points sont relativement bien regroupés autour des centroïdes.

8. Evaluation du modèle en utilisant model.score(X) :

- Code utilisé :

```
#8)

# Évaluation du modèle
score = model.score(X)
print("Score du modèle:", score)
```

- Résultat obtenu :

```
Coût (inertia): 168.72072611999806
Score du modèle: -168.720726119998
```

On remarque que la méthode model.score(X) retourne l'opposé de la valeur de l'inertie du modèle. En d'autres termes, model.score(X) est égal à -model.inertia_.

Cela signifie que le score est simplement une autre manière de représenter le coût du modèle, mais avec un signe négatif.

9. Commentaires :

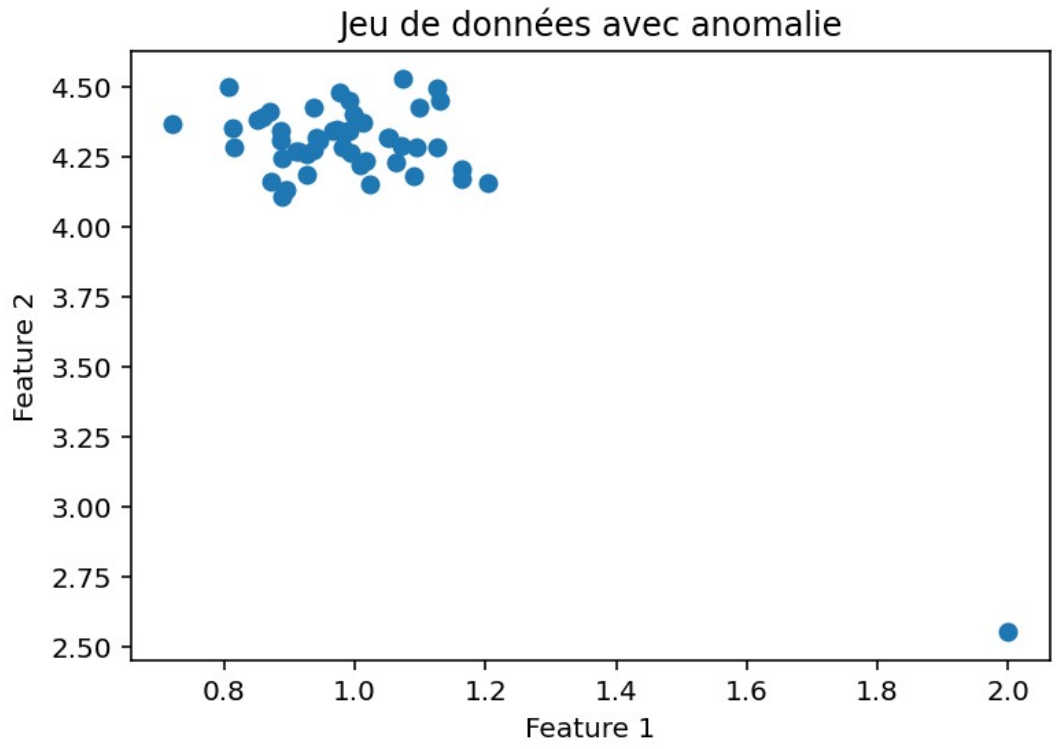
```
#9)

# Initialisation d'une liste vide pour stocker les valeurs d'inertia
inertia = []

# Définition de la plage de valeurs pour le nombre de clusters à tester
K_range = range(1, 20)

# Boucle sur chaque valeur de k dans la plage définie
for k in K_range:
    # Création et entraînement du modèle KMeans avec k clusters
    model = KMeans(n_clusters=k).fit(X)

    # Ajout de l'inertia du modèle à la liste
    inertia.append(model.inertia_)

# Tracé de la courbe de l'inertia en fonction du nombre de clusters
plt.plot(K_range, inertia)

# Ajout d'une étiquette pour l'axe des abscisses
plt.xlabel('Nombre de clusters')

# Ajout d'une étiquette pour l'axe des ordonnées
plt.ylabel('Coût du modèle (Inertia)')

# Ajout d'un titre au graphique
plt.title('Méthode du coude')

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

Exercice 2 : Détection d'anomalie (Isolation Forest algorithm)

1. Génération et affichage du jeu de données :

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import numpy as np

# Génération du jeu de données
X, y = make_blobs(n_samples=50, centers=1, cluster_std=0.1, random_state=0)

# Ajout d'un point anomalie
X[-1, :] = np.array([2, 2.55])

# Affichage du jeu de données
plt.scatter(X[:, 0], X[:, 1])
plt.title("Jeu de données avec anomalie")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

Affichage du jeu de donnée :



2. Les codes utilisés pour cette partie :

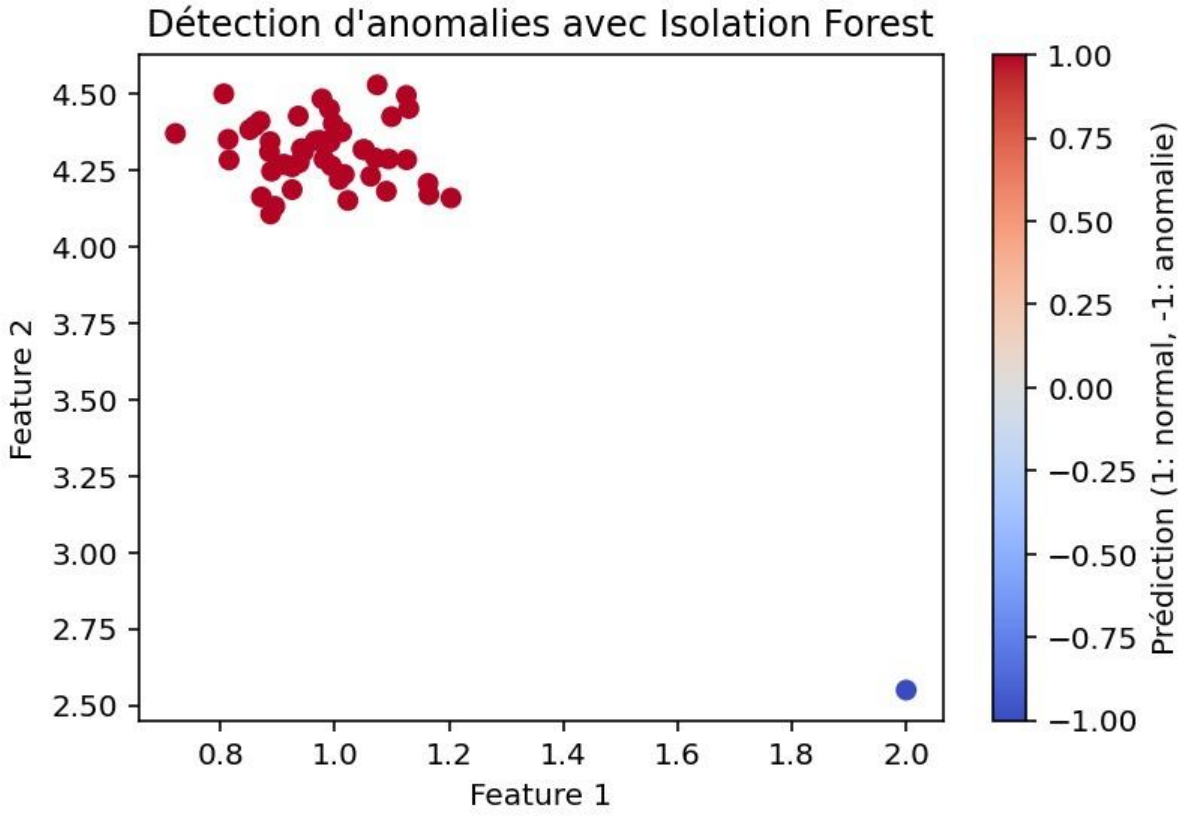
Pour la détection d'anomalies avec isolation Forest :

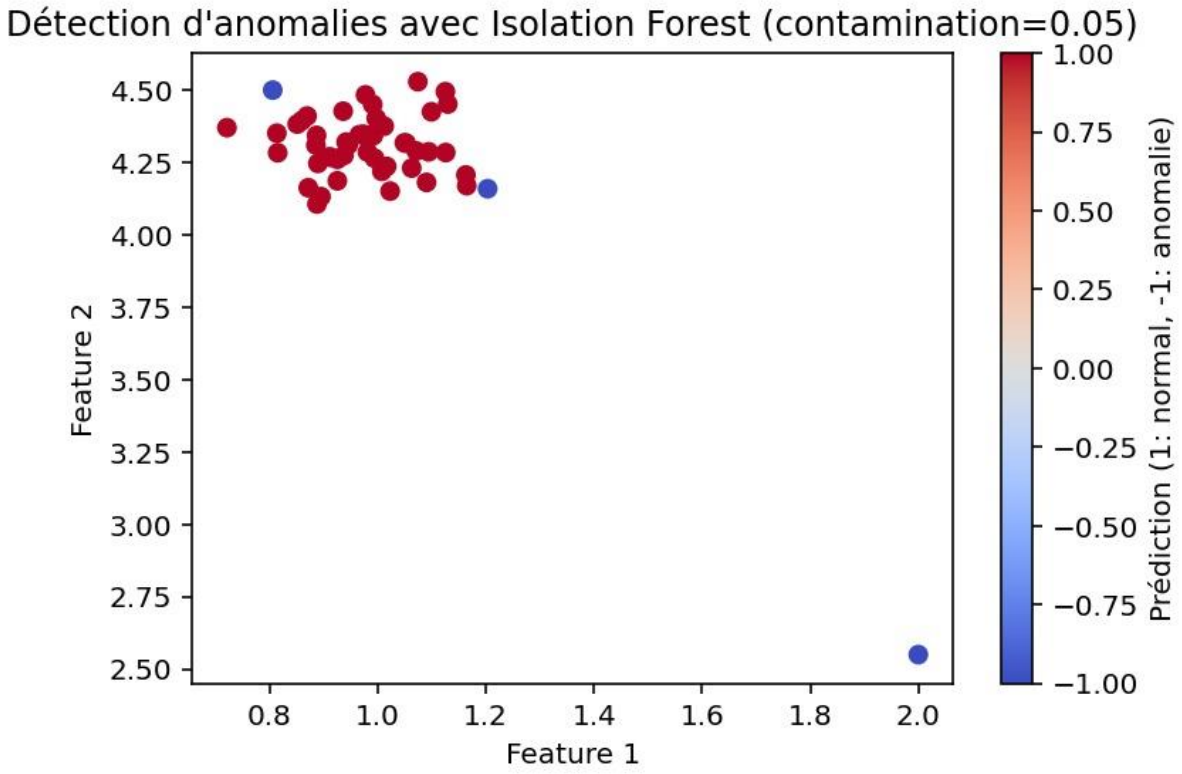
```python
# Affichage des résultats
plt.scatter(X[:, 0], X[:, 1], c=predictions, cmap='coolwarm')
plt.title("Détection d'anomalies avec Isolation Forest")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.colorbar(label='Prédiction (1: normal, -1: anomalie)')
plt.show()
```

Pour la détection d'anomalies avec isolation Forest (contamination =0.05)

```python
# Modification de l'hyperparamètre contamination
model = IsolationForest(contamination=0.05)

# Entraînement du modèle
model.fit(X)

# Prédiction des anomalies
predictions = model.predict(X)

# Affichage des résultats
plt.scatter(X[:, 0], X[:, 1], c=predictions, cmap='coolwarm')
plt.title("Détection d'anomalies avec Isolation Forest (contamination=0.05)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.colorbar(label='Prédiction (1: normal, -1: anomalie)')
plt.show()
```

Résultats obtenus :





la contamination=0.01 représente la proportion estimée d'anomalies dans le jeu de données. Ici, cela signifie que l'on s'attend à ce que 1% des données soient des anomalies.

En augmentant la valeur de contamination, le modèle s'attend à une proportion plus élevée d'anomalies dans le jeu de données. Cela peut conduire à une détection plus agressive des anomalies.

Exercice 3 : Détection des chiffres manuscrits mal écrits dans la base « digits » (Isolation Forest algorithm)

1. Importation de la librairie pour télécharger le jeu de données :

```
#1)
from sklearn.datasets import load_digits
```

2. Telechargement du jeu de données correspondant à des chiffres manuscrits representés par des images :

```
#2)
# Téléchargement du jeu de données
digits = load_digits()

# Récupération des images et des targets
images = digits.images
```

Le téléchargement du jeu de données "digits" permet d'obtenir une collection de chiffres manuscrits représentés sous forme d'images, facilitant ainsi l'analyse et la détection des anomalies dans les écritures.

3. Récupération des images et les targets :

```
#3)
X = digits.data
y = digits.target
```

La récupération des images et des targets permet de séparer les données en caractéristiques (images des chiffres) et en étiquettes (valeurs des chiffres), facilitant ainsi l'analyse et le traitement des données.

4. La taille de X et des images :

```
# Taille de X
print("Taille de X:", X.shape)

# Taille de chaque image
print("Taille de chaque image:", images.shape[1:])
```

Tailles obtenues :

```
Taille de X: (1797, 64)
Taille de chaque image: (8, 8)
```

La taille de X (1797, 64) indique qu'il y a 1797 images de chiffres manuscrits, chacune représentée par 64 caractéristiques.

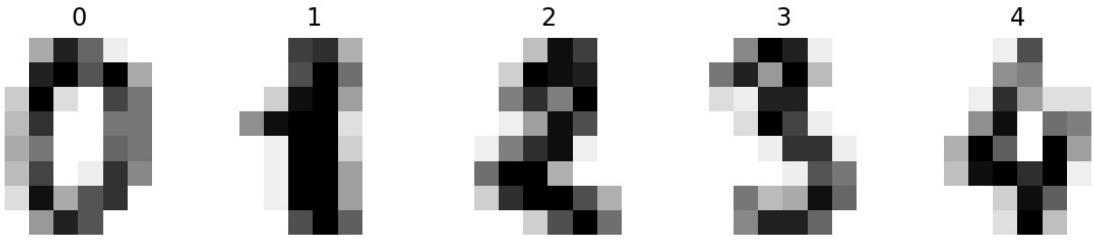
La taille de chaque image (8, 8) signifie que chaque image est une grille de 8x8 pixels, aplatie en un vecteur de 64 caractéristiques pour l'analyse.

Visualisation de quelques images :

- Code utilisé pour permettre ca :

```
import matplotlib.pyplot as plt

# Visualisation de quelques images
fig, axes = plt.subplots(1, 5, figsize=(10, 3))
for ax, image, label in zip(axes, images[0:5], y[0:5]):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)
plt.show()
```

- Images obtenues :



5. Sélection du modèle IsolationForest :

```
from sklearn.ensemble import IsolationForest

# Sélection du modèle IsolationForest
model = IsolationForest(random_state=0, contamination=0.02)
```

6. Entrainement du modèle :

```
#6)
# Entraînement du modèle
model.fit(X)
```

7. Evaluation du modèle :

```
#7)
# Évaluation du modèle
predictions = model.predict(X)
```

- Interprétation :

L'évaluation du modèle avec model.predict(X) permet de déterminer quels chiffres manuscrits sont considérés comme des anomalies, indiquant ainsi les chiffres potentiellement mal écrits

8. Filtrage des prédictions égales a –1 :

```
# Interprétation des résultats
outliers = predictions == -1
print("Nombre d'anomalies détectées:", outliers.sum())
```

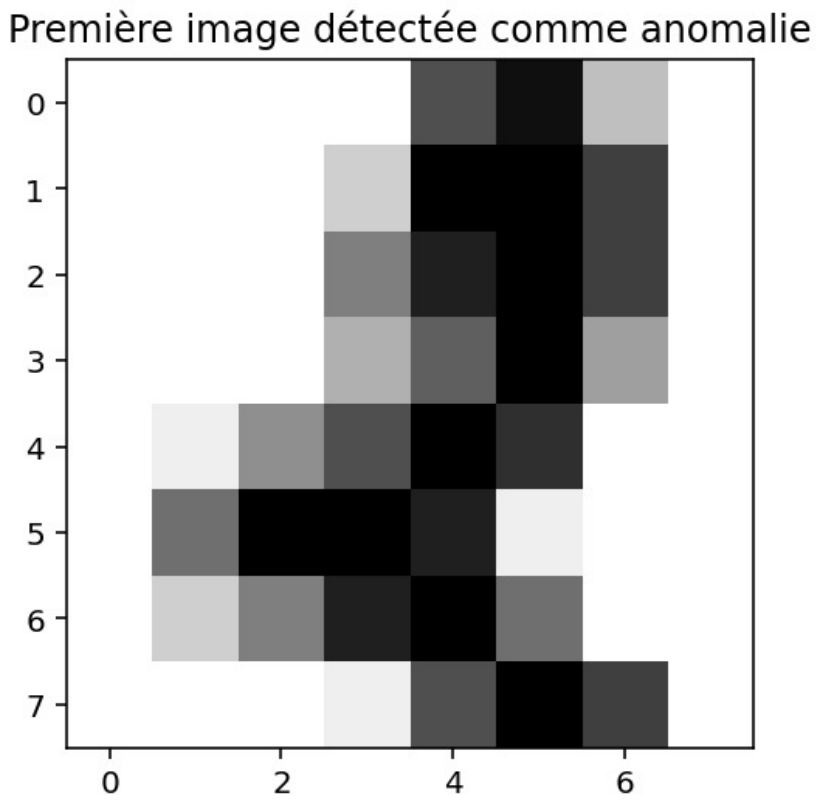
On a trouvé 5 anomalies.

9. Utilisation de Images[outliers][0] et affichage de l'image :

- Code utilisé :

```
# Sélectionnez la première image détectée comme anomalie
first_outlier_image = images[outliers][0]

# Affichage de la première image détectée comme anomalie
plt.imshow(first_outlier_image, cmap=plt.cm.gray_r, interpolation='nearest')
plt.title("Première image détectée comme anomalie")
plt.show()
```

- Résultat :



L'image a été classée comme anomalie à cause de la qualité d'écriture, la forme et la structure du chiffre est difficile à discerner.

10. Modification de l'hyperparamètre contamination :

```
#10/
from sklearn.ensemble import IsolationForest

# Modification de l'hyperparamètre contamination
model = IsolationForest(random_state=0, contamination=0.05)

# Entraînement du modèle
model.fit(X)

# Évaluation du modèle
predictions = model.predict(X)

# Interprétation des résultats
outliers = predictions == -1
print("Nombre d'anomalies détectées avec contamination=0.05:", outliers.sum())
```

- Résultat :

```
Nombre d'anomalies détectées avec contamination=0.05: 90
```

En augmentant la valeur de contamination à 0.05, le modèle s'attend à une proportion plus élevée d'anomalies dans le jeu de données. Cela est conduit à une détection plus agressive des anomalies, c'est-à-dire que plus de points seront classés comme anomalies, comme il est marqué dans le résultat.

Exercice 4 : Réduction de la dimensionnalité

A. Téléchargement du jeu de données “Digits”:

1. Importation des librariries :

```
from sklearn.datasets import load_digits
```

2. Et
3. Téléchargement du jeu de données et récuperation des images et les targets :

```
# Téléchargement du jeu de données
digits = load_digits()

# Récupération des images et des targets
images = digits.images
X = digits.data
y = digits.target
```

4. Dimensions du jeu de données :

- Code utilisé :

```
# Affichage des dimensions du jeu de données
print("Dimensions du jeu de données X:", X.shape)
print("Dimensions des cibles y:", y.shape)
```

- Résultat :

```
Dimensions du jeu de données X: (1797, 64)
Dimensions des cibles y: (1797,)
```

X.shape nous donne le nombre de samples (1797) et le nombre de features (64) pour chaque sample et y.shape nous donne le nombre de cibles (1797), chaque cible correspondant à un chiffre manuscrit.

B. Visualisation des données :

1.

```
#B
from sklearn.decomposition import PCA
```

2. selection du modele PCA et fixer à 2 le nombre de dimension :

```
# Sélection du modèle PCA avec 2 composantes
model = PCA(n_components=2)
```

3. entrainement du modele :

```
# Entraînement du modèle et réduction des données
X_reduced = model.fit_transform(X)
```

4. Vérification de la dimension du tableau X-reduced :

- code utilisé :

```
# Affichage des dimensions des données réduites
print("Dimensions des données réduites X_reduced:", X_reduced.shape)
```

- résultat obtenu :

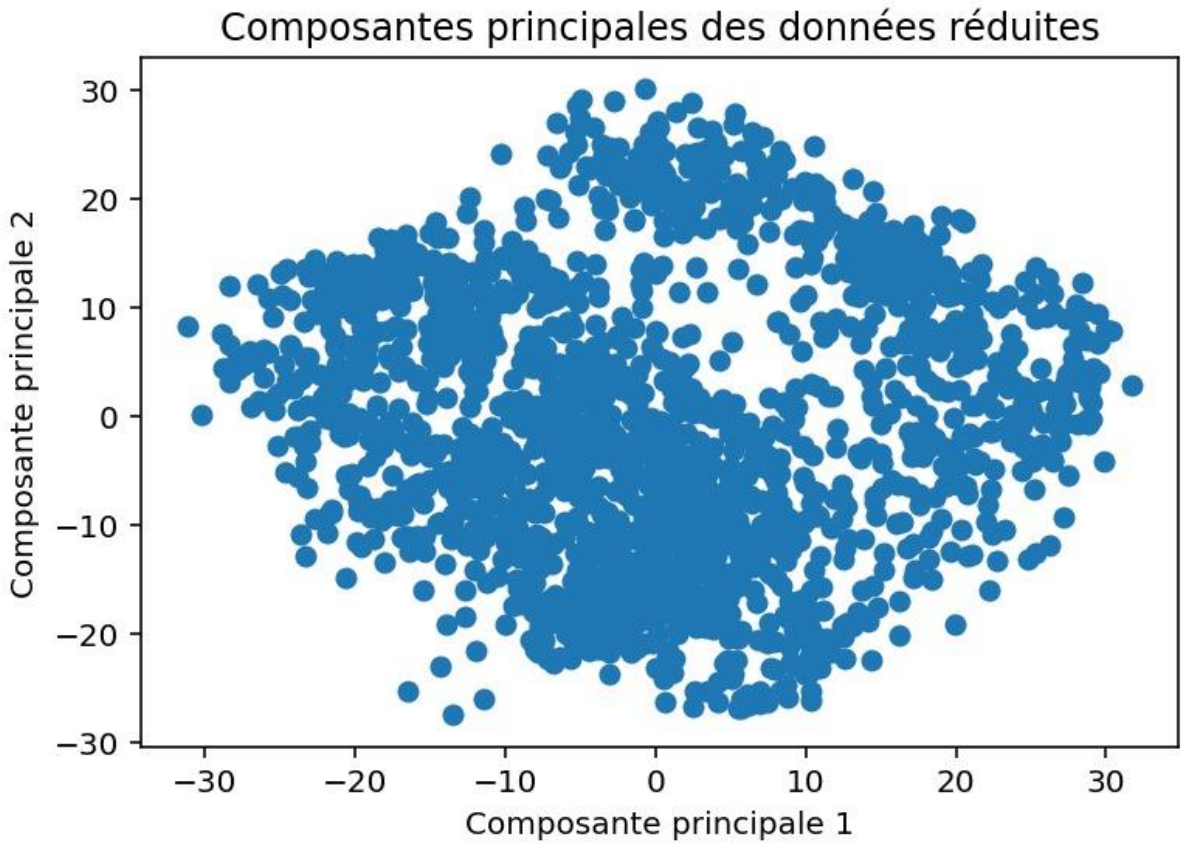
```
Dimensions des données réduites X_reduced: (1797, 2)
```

X_reduced.shape nous donne le nombre de samples (1797) et le nombre de composantes principales (2) après la réduction de dimensionnalité.

5. Observation des composants de X-reduced :

```
import matplotlib.pyplot as plt

# Affichage des composantes principales
plt.scatter(X_reduced[:, 0], X_reduced[:, 1])
plt.title("Composantes principales des données réduites")
plt.xlabel("Composante principale 1")
plt.ylabel("Composante principale 2")
plt.show()
```

Résultat :



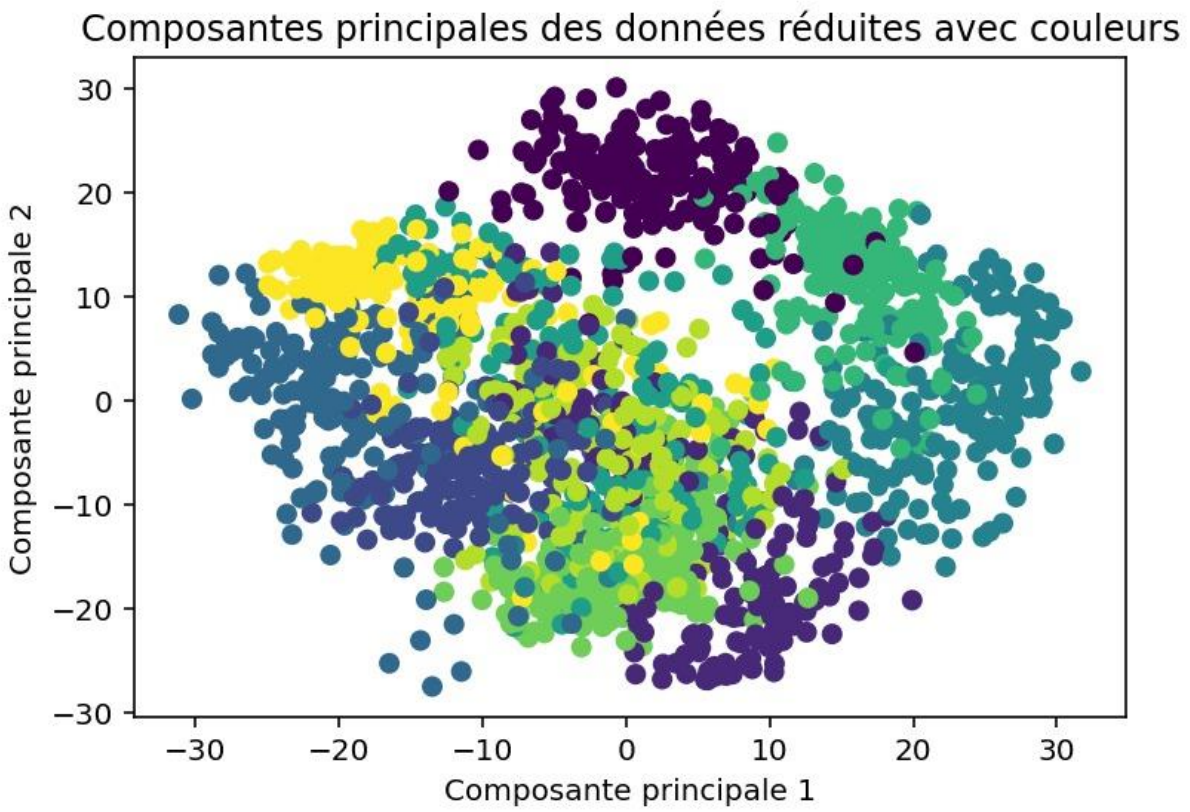
- Interpretation :

Ce graphique montre la réduction des données en deux dimensions principales, révélant une concentration centrale des points avec une dispersion vers les bords, ce qui aide à visualiser la structure et les anomalies des données.

6. L'ajout des couleurs sur le graphique :

```
# Ajout de couleurs au graphique en fonction des cibles
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, cmap='viridis')
plt.title("Composantes principales des données réduites avec couleurs")
plt.xlabel("Composante principale 1")
plt.ylabel("Composante principale 2")
plt.show()
```

- resultat :



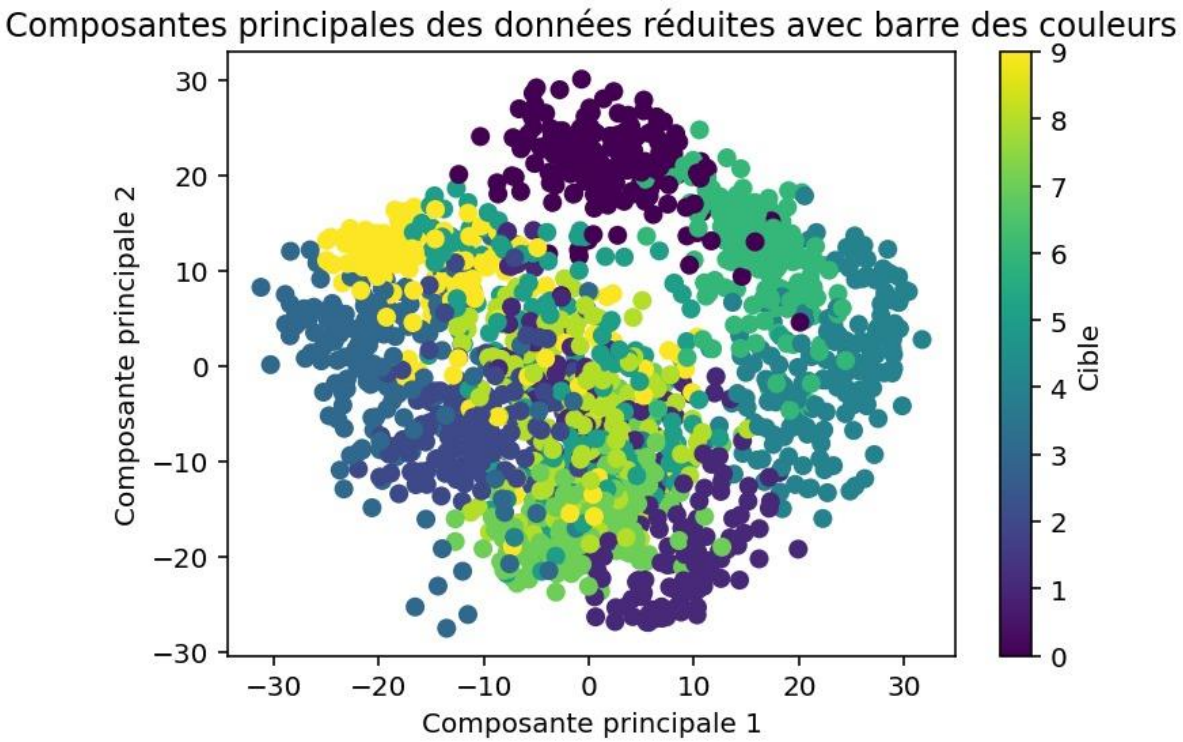
y contient les cibles, c'est-à-dire les labels des chiffres manuscrits (de 0 à 9).

7. L'ajout d'une barre des couleurs pour analyser la visualisation :

- Code utilisé :

```
# Ajout d'une barre des couleurs
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, cmap='viridis')
plt.colorbar(label='Cible')
plt.title("Composantes principales des données réduites avec barre des couleurs")
plt.xlabel("Composante principale 1")
plt.ylabel("Composante principale 2")
plt.show()
```

Resultat :



8. Interpretation :

La visualisation montre les données réduites à deux dimensions en utilisant PCA.

Chaque point représente un chiffre manuscrit, et les couleurs indiquent les différentes cibles (chiffres de 0 à 9).

Les points de même couleur tendent à se regrouper, ce qui montre que les chiffres similaires sont projetés près les uns des autres dans l'espace des composantes principales. Comme visualisé dans le graphe d'ACP.

9. Les axes représentent les deux premières composantes principales. La composante principale 1 (axe des abscisses) et la composante principale 2 (axe des ordonnées) sont les directions dans lesquelles la variance des données est maximisée.

10. Analyse du contenu de chaque composante :

```
# Affichage des composantes du modèle PCA
print("Forme des composantes du modèle PCA:", model.components_.shape)
```

Resultat :

```
Forme des composantes du modèle PCA: (2, 64)
```

model.components_.shape nous donne la forme des composantes du modèle PCA, qui est (2, 64). Cela signifie qu'il y a 2 composantes principales, chacune étant une combinaison linéaire des 64 features originales.

C. Compression de données :

1. Entrainement du model sur le même nombre de dimension que X :

```
# Sélection du modèle PCA avec le même nombre de dimensions que X
model = PCA(n_components=X.shape[1])
```

2. Entrainement du model :

```
# Entraînement du modèle et réduction des données
X_reduced = model.fit_transform(X)
```

3. Pourcentage de variance :

- Code utilisé :

```
# Affichage du pourcentage de variance expliquée par chaque composante
explained_variance_ratio = model.explained_variance_ratio_
print("Pourcentage de variance expliquée par chaque composante:", explained_variance_ratio)
```

- Résultat :

```
Pourcentage de variance expliquée par chaque composante: [1.48905936e-01
1.36187712e-01 1.17945938e-01 8.40997942e-02
 5.78241466e-02 4.91691032e-02 4.31598701e-02 3.66137258e-02
 3.35324810e-02 3.07880621e-02 2.37234084e-02 2.27269657e-02
 1.82186331e-02 1.77385494e-02 1.46710109e-02 1.40971560e-02
 1.31858920e-02 1.24813782e-02 1.01771796e-02 9.05617439e-03
 8.89538461e-03 7.97123157e-03 7.67493255e-03 7.22903569e-03
 6.95888851e-03 5.96081458e-03 5.75614688e-03 5.15157582e-03
 4.89539777e-03 4.28887968e-03 3.73606048e-03 3.53274223e-03
 3.36683986e-03 3.28029851e-03 3.08320884e-03 2.93778629e-03
 2.56588609e-03 2.27742397e-03 2.22277922e-03 2.11430393e-03
 1.89909062e-03 1.58652907e-03 1.51159934e-03 1.40578764e-03
 1.16622290e-03 1.07492521e-03 9.64053065e-04 7.74630271e-04
 5.57211553e-04 4.04330693e-04 2.09916327e-04 8.24797098e-05
 5.25149980e-05 5.05243719e-05 3.29961363e-05 1.24365445e-05
 7.04827911e-06 3.01432139e-06 1.06230800e-06 5.50074587e-07
 3.42905702e-07 9.50687638e-34 9.50687638e-34 9.36179501e-34]
```

4. Analyse de la somme cumulée de variance :

Code utilisé :

```python
import numpy as np

# Calcul de la somme cumulée de variance
cumulative_variance = np.cumsum(explained_variance_ratio)
print("Somme cumulée de variance:", cumulative_variance)
```

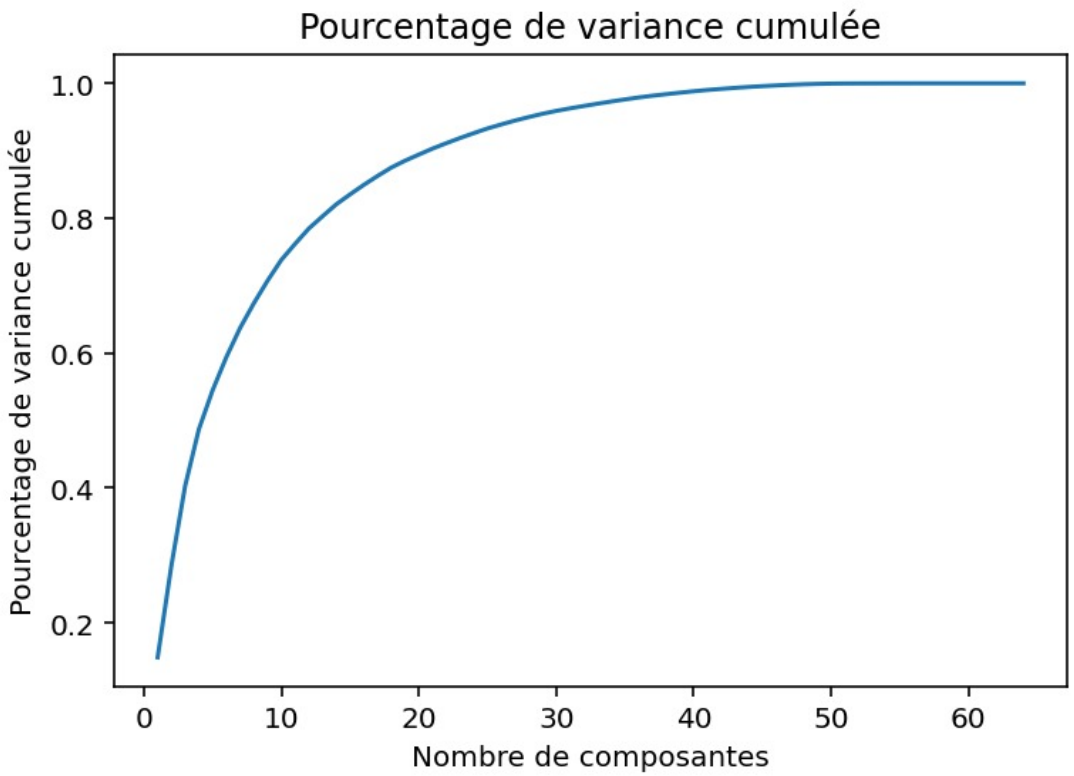
Résultat :

```
Somme cumulée de variance: [0.14890594 0.28509365 0.40303959 0.48713938 0.54496353
0.59413263
 0.6372925  0.67390623 0.70743871 0.73822677 0.76195018 0.78467714
 0.80289578 0.82063433 0.83530534 0.84940249 0.86258838 0.87506976
 0.88524694 0.89430312 0.9031985  0.91116973 0.91884467 0.9260737
 0.93303259 0.9389934  0.94474955 0.94990113 0.95479652 0.9590854
 0.96282146 0.96635421 0.96972105 0.97300135 0.97608455 0.97902234
 0.98158823 0.98386565 0.98608843 0.98820273 0.99010182 0.99168835
 0.99319995 0.99460574 0.99577196 0.99684689 0.99781094 0.99858557
 0.99914278 0.99954711 0.99975703 0.99983951 0.99989203 0.99994255
 0.99997555 0.99998798 0.99999503 0.99999804 0.99999911 0.99999966
 1.         1.         1.         1.        ]
```

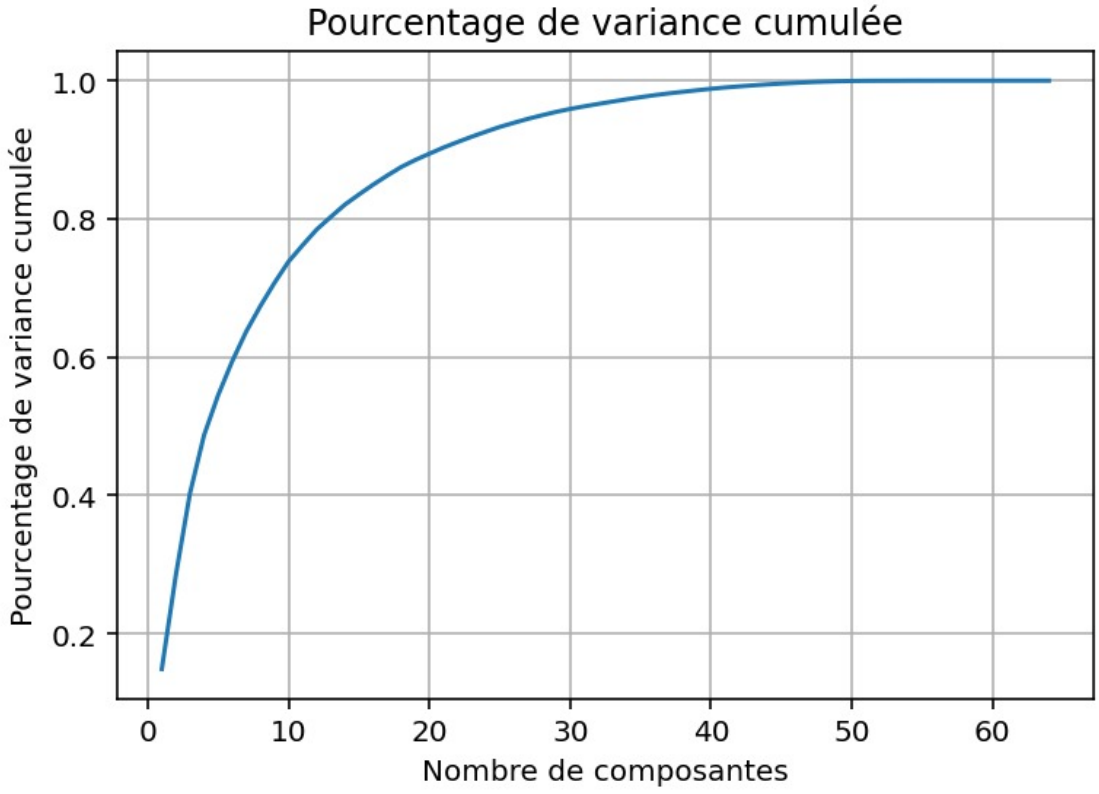
5. Pourcentage de variance cumulé :

```python
# Tracé du pourcentage de variance cumulée
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance)
plt.title("Pourcentage de variance cumulée")
plt.xlabel("Nombre de composantes")
plt.ylabel("Pourcentage de variance cumulée")
plt.show()
```

Résultat :



6. 90% de variance cumulée :



Pour mieux interpréter le nombre de composantes on applique le grid sur la courbe, on interprete qu'a partir de 20 composantes le graphique montre que 90% de variance (0,9)

7. Code utilisé :

```
# Détermination du nombre de composantes pour atteindre 99% de variance
n_components_99 = np.argmax(cumulative_variance >= 0.99) + 1
print("Nombre de composantes pour atteindre 99% de variance:", n_components_99)
```

résultat :

```
Nombre de composantes pour atteindre 99% de variance: 41
```

Interpretation :

Le résultat indique que 41 composantes sont nécessaires pour atteindre 99% de la variance expliquée dans le jeu de données analysé.

8. Entrainement du modèle PCA :

```
# Sélection du modèle PCA avec le nombre de composantes pour atteindre 99% de variance
model = PCA(n_components=n_components_99)

# Entraînement du modèle et réduction des données
X_reduced = model.fit_transform(X)
```

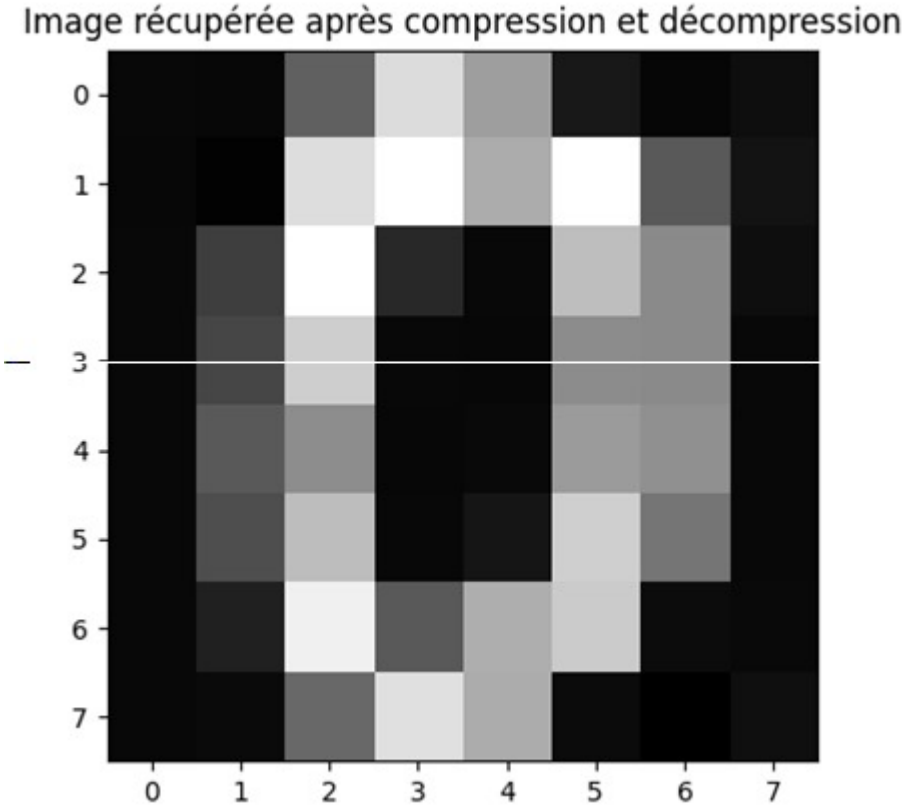
9. Décomposition des images :

```
# Décompression des données
X_recovered = model.inverse_transform(X_reduced)
```

10. Affichage des images de X-recoverder :

```
# Affichage d'une image décompressée
plt.imshow(X_recovered[0].reshape((8, 8)), cmap=plt.cm.gray_r, interpolation='nearest')
plt.title("Image décompressée")
plt.show()
```

- Image affichée :



L'image décompressée semble représenter le chiffre 0, mais la forme et la structure ne sont pas très claires.

On a utilisé le nombre des composantes à 99% de la variance.

11.
```python
from sklearn.preprocessing import MinMaxScaler

# Mise à échelle des données
scaler = MinMaxScaler()
data_rescaled = scaler.fit_transform(X)

# Sélection du modèle PCA avec 40% de variance préservée
model = PCA(n_components=0.40)

# Entraînement du modèle et réduction des données
model.fit(data_rescaled)
X_reduced = model.transform(data_rescaled)

# Vérification du nombre de composantes utilisées
print("Nombre de composantes utilisées pour 40% de variance:", model.n_components_)
```

Resultat :

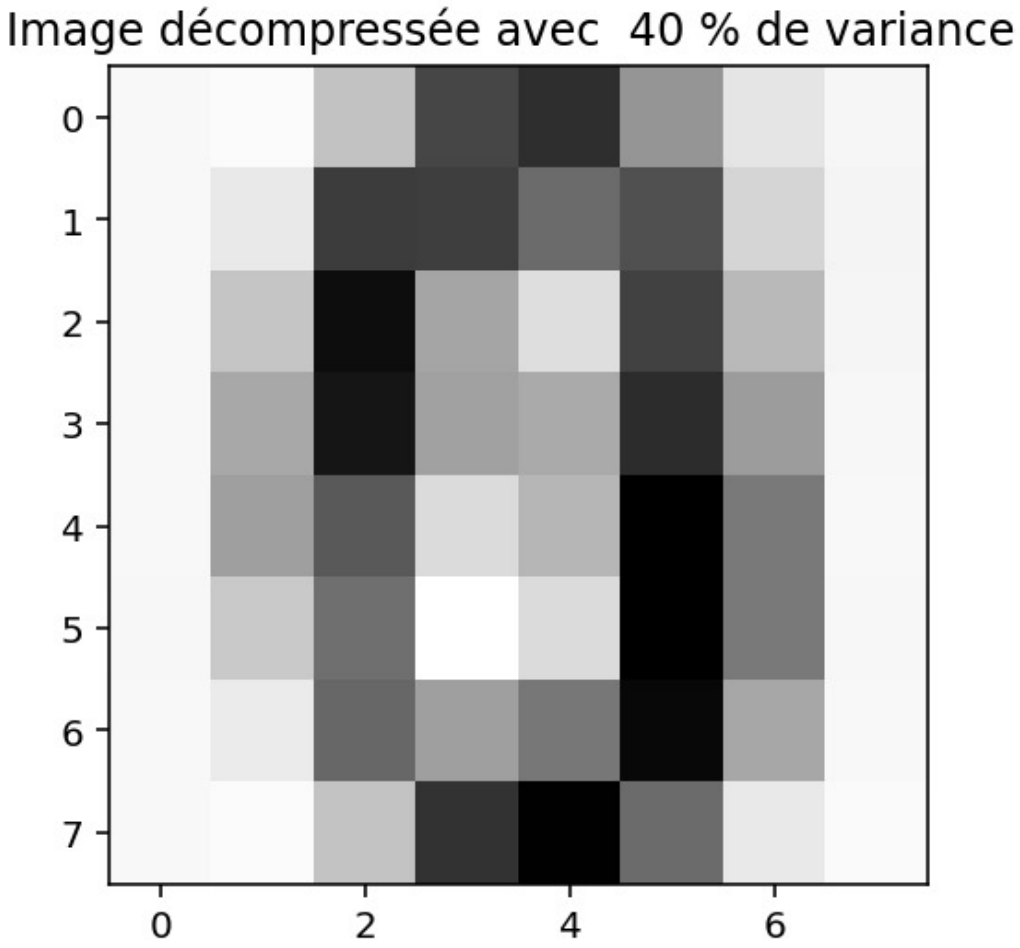
```
Nombre de composantes utilisées pour 40% de variance: 3
```

12. Analyse de l’impact de la réduction du nombre de composante surla qualité des images :

Code utilisé :

```
X_recovered2 = model.inverse_transform(X_reduced)
plt.imshow(X_recovered2[0].reshape((8, 8)), cmap=plt.cm.gray_r, interpolation='nearest')
plt.title("Image décompressée avec  40 % de variance ")
plt.show()
```

Résultat :



On voit bien que l'image n’est pas du tout claire et on ne peut pas distinguer le chiffre à cause de conservation de 40 % de variance, l'image est floue ce qui indique que la réduction de dimensionnalité a perdu certaines informations importantes.

Conclusion :

En conclusion, ce travail pratique nous a permis de mettre en application les concepts théoriques de l'apprentissage non supervisé à travers des exercices concrets de clustering, de détection d'anomalies et de réduction de dimensionnalité.

L'utilisation de l'algorithme K-means pour le clustering a démontré son efficacité à regrouper des données en clusters distincts, tandis que l'algorithme Isolation Forest s'est révélé être un outil puissant pour la détection d'anomalies dans des jeux de données variés.

Enfin, l'Analyse en Composantes Principales (ACP) a permis de réduire la dimensionnalité des données tout en conservant une grande partie de la variance, facilitant ainsi la visualisation et l'interprétation des données.