BASKARAN

Amalan

N°12001105

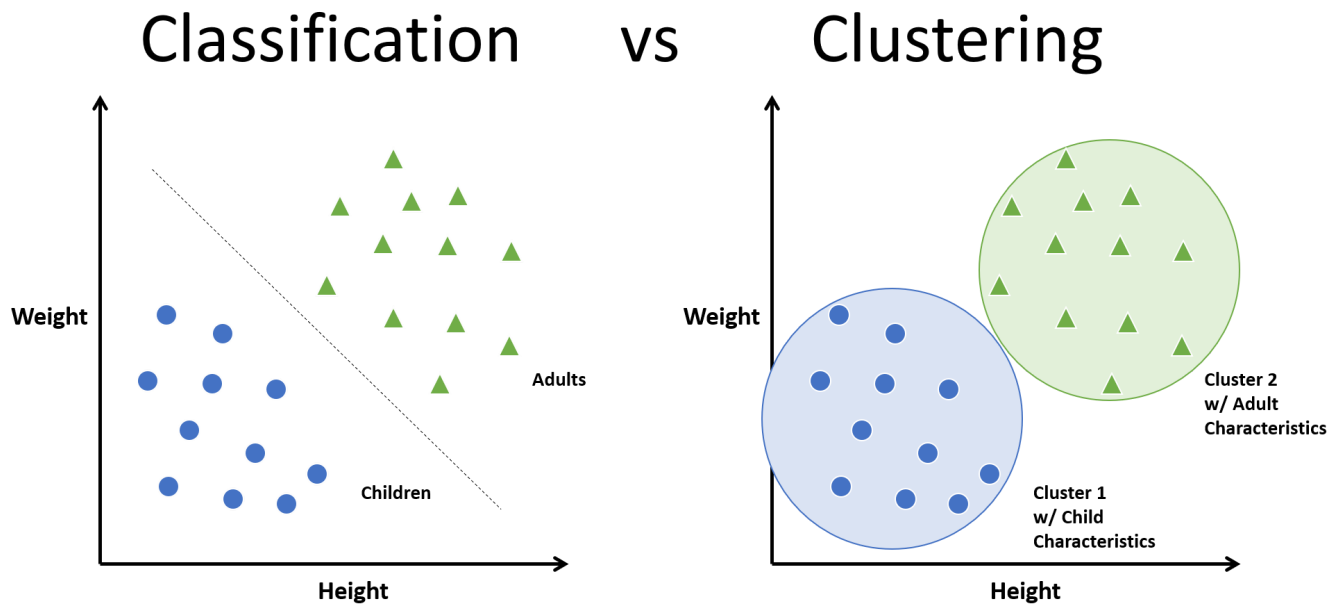
19/11/2023

---------- TP 3 : Apprentissage Non-Supervisé ----------

Entrée [464]:

```
import pandas as pd # Toutes les libraries utlilsé au cours de ce TP.
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.ensemble import IsolationForest
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

---------- EXO 1 : Clustering (K-means clustering) ----------



Supervisé vs Non Supervisé

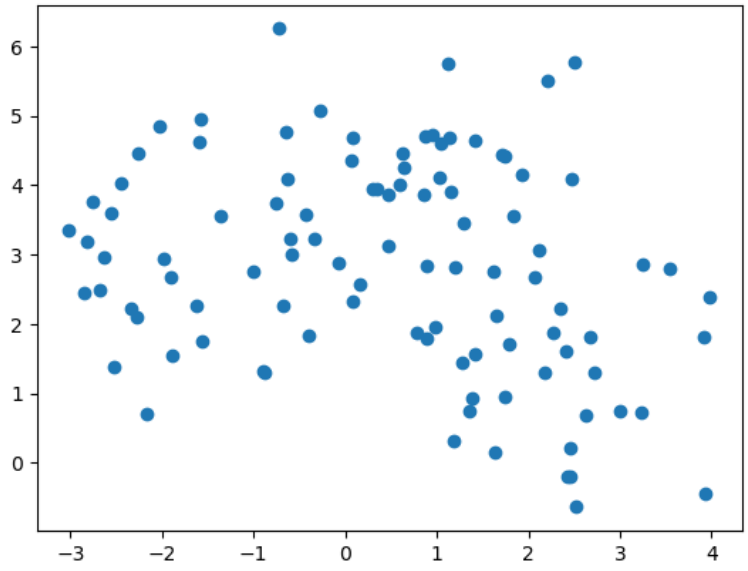
Dans ce TP nous allons apprendre les techniques de machine learning non supervisé

Entrée [465]:

```
X,y = make_blobs(n_samples = 100,centers = 3,n_features = 2, random_state = 0) # génération des données
plt.scatter(X[:,0],X[:,1])
```

Out[465]:

```
<matplotlib.collections.PathCollection at 0x2225d2c7310>
```



Entrée [466]:

```
X_train,X_test,Y_train,Y_test = train_test_split(X,y,test_size = 0.2) # Séparation pour Train et test
```

Entrée [467]:

```
model = KMeans(n_clusters = 3,n_init="auto") # model
```

- la signification de n_clusterq = 3 est le nombre de clusters à former ainsi que le nombre de centroïdes à générer.

- le nombre d'initialisation fixé par défaut est 10 mais j'utlise auto

- le nombre d'itérations maximal utilisé par défaut est 300

- la méthode d'initialisation utilisée est lloyd

- comment sont choisis les points des centres ?

calculé comme la moyenne des points appartenant au même cluster

Entrée [ ]:

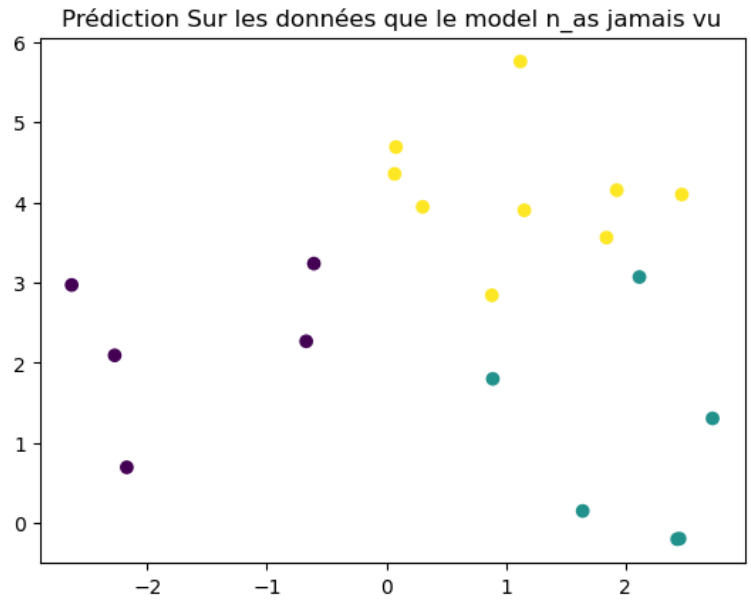
```
model.fit(X_train)
```

Entrée [469]:

```
model.predict(X_test)
plt.scatter(X_test[:,0],X_test[:,1],c=model.predict(X_test))
plt.title('Prédiction Sur les données que le model n_as jamais vu')
```

Out[469]:

```
Text(0.5, 1.0, 'Prédiction Sur les données que le model n_as jamais vu')
```

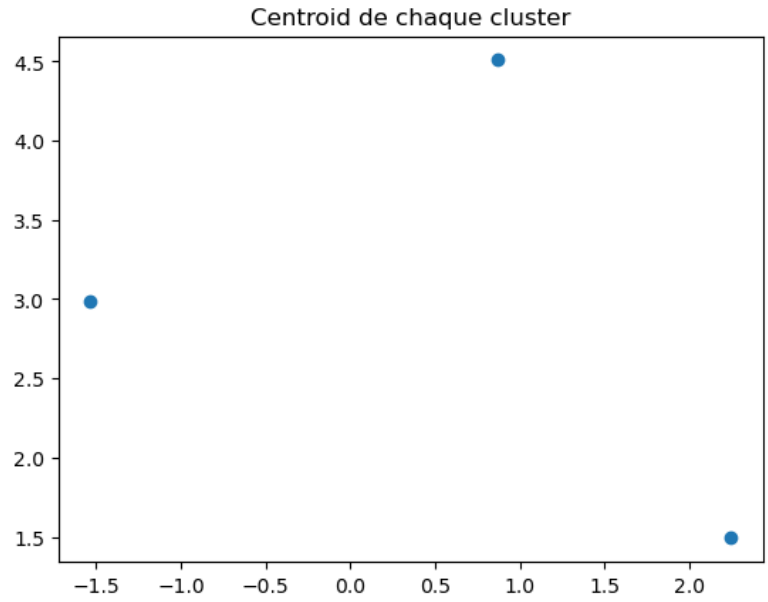


Entrée [470]:

```
model.cluster_centers_
model.cluster_centers_[:,0],model.cluster_centers_[:,1]
plt.scatter(model.cluster_centers_[:,0],model.cluster_centers_[:,1])
plt.title('Centroid de chaque cluster')
```

Out[470]:

```
Text(0.5, 1.0, 'Centroid de chaque cluster')
```



6) A quoi servent ces centroides ?

Ils peuvent être utilisés pour mesurer la distance entre les points et les clusters, ou pour visualiser les clusters dans un espace à faible dimension ou encore lores de la prediction pour prédire les données appartenants à quel cluster.

7) Calculez le coût engendré par ce modèle en utilisant model.inertia_. Commet est-il calculé ?

Somme des carrés des distances des échantillons jusqu'à leur centre de cluster le plus proche

Entrée [471]:

```
model.inertia_
```

Out[471]:

```
133.92651641676642
```

8) Evaluez le modèle en utilisant cette fois-ci model.score(X). Comparez le résultat obtenu à la question précédente.

Entrée [472]:

```
model.score(X_train)
```

Out[472]:

```
-133.92651641676642
```

On peut voir que le model.score est un nombre négative donc on utlisera inertia pour avoir la même valeur positif.

model.inertia = abs(model.score())

Entrée [ ]:

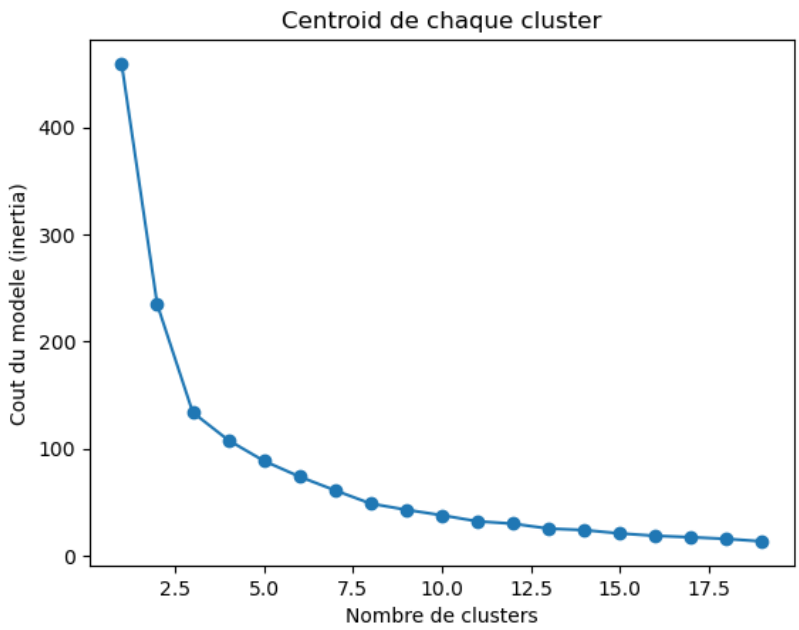
```
inertia = [] #Tableau
K_range = range(1,20)
for k in K_range:
    model = KMeans(n_clusters = k).fit(X_train) # entrainement avec n_cluster 1 à 20
    inertia.append(model.inertia_) # on prend inertia de chaque n_cluster et on met dans ce tableau.
```

Entrée [474]:

```
plt.plot(K_range,inertia) # les inertia en fonction de n_clusters
plt.scatter(K_range,inertia)
plt.xlabel('Nombre de clusters')
plt.ylabel('Cout du modele (inertia)')
plt.title('Centroid de chaque cluster')
```

Out[474]:

```
Text(0.5, 1.0, 'Centroid de chaque cluster')
```



« The Elbow method » ou la méthode du coude L'emplacement d'une courbe (coude) dans le graphique est généralement considéré comme un indicateur du nombre approprié de groupes. Les distances ne varient plus rapidement après cette inflexion, c'est-à-dire qu'une augmentation de k n'aura pas un grand effet sur les distances entre les points de données et leurs centroïdes respectives.

K = 4 ou 5 est le meilleur approximation de K

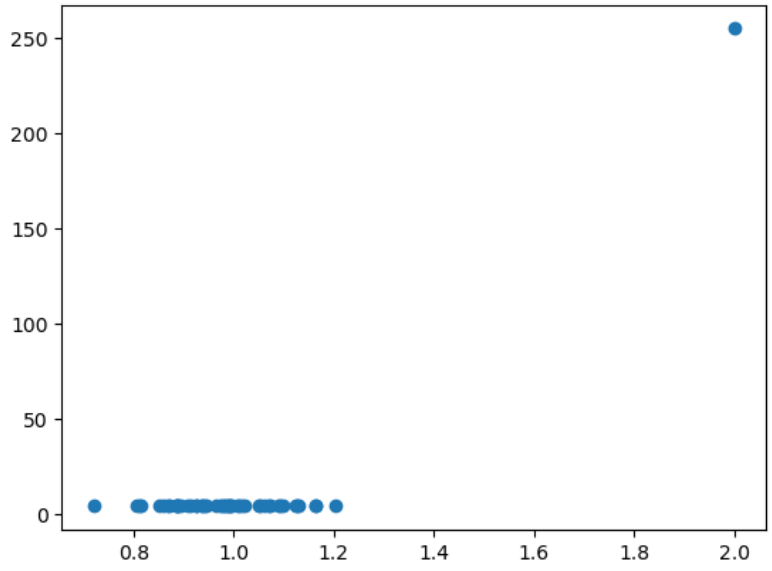
EXO 2 : Détection d'anomalie

Entrée [475]:

```
X,y = make_blobs(n_samples = 50,centers = 1,cluster_std = 0.1, random_state = 0) # crée jeu données avec un point éloigné.
X[-1,:] = np.array([2,255])
plt.scatter(X[:,0],X[:,1])
```

Out[475]:

```
<matplotlib.collections.PathCollection at 0x2225d9084d0>
```



Entrée [476]:

```
model = IsolationForest(contamination=0.01)
```

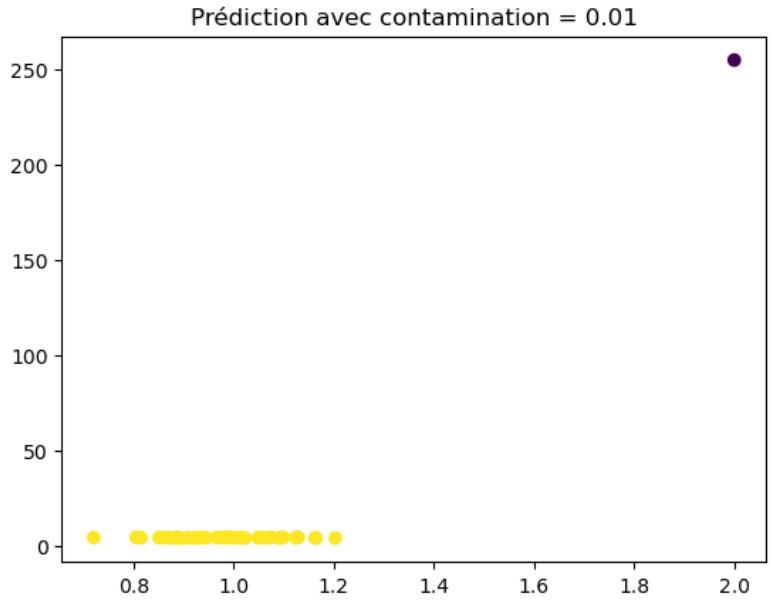
contamination est le seuil des points à considérer comme anormaux est défini par le taux de contamination. Dans notre exemple, le taux de contamination est égal à 0,01 (soit 1%). Ainsi, on considère que nos données contiennent 1% d'anomalies.

Entrée [477]:

```
model.fit(X,y)
plt.scatter(X[:,0],X[:,1],c= model.predict(X))
plt.title('Prédiction avec contamination = 0.01')
```

Out[477]:

```
Text(0.5, 1.0, 'Prédiction avec contamination = 0.01')
```

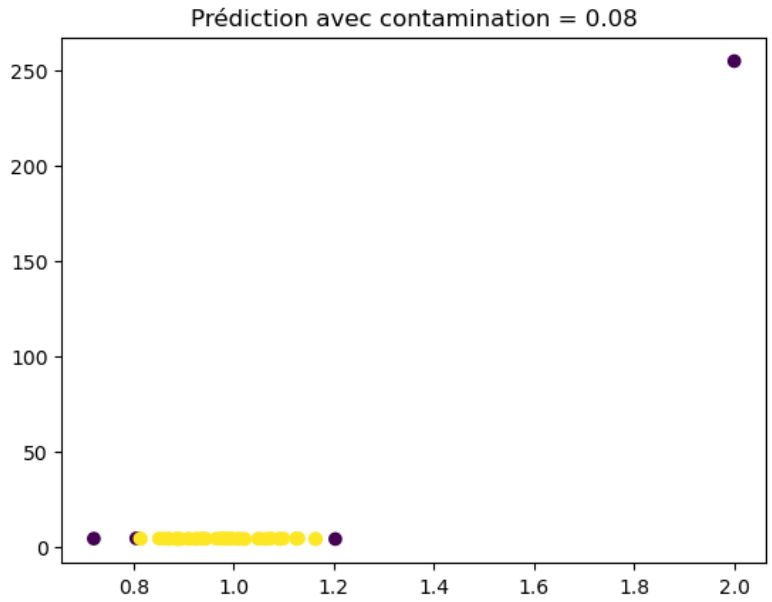


Entrée [478]:

```
model = IsolationForest(contamination=0.08)
model.fit(X,y)
plt.scatter(X[:,0],X[:,1],c= model.predict(X))
plt.title('Prédiction avec contamination = 0.08')
```

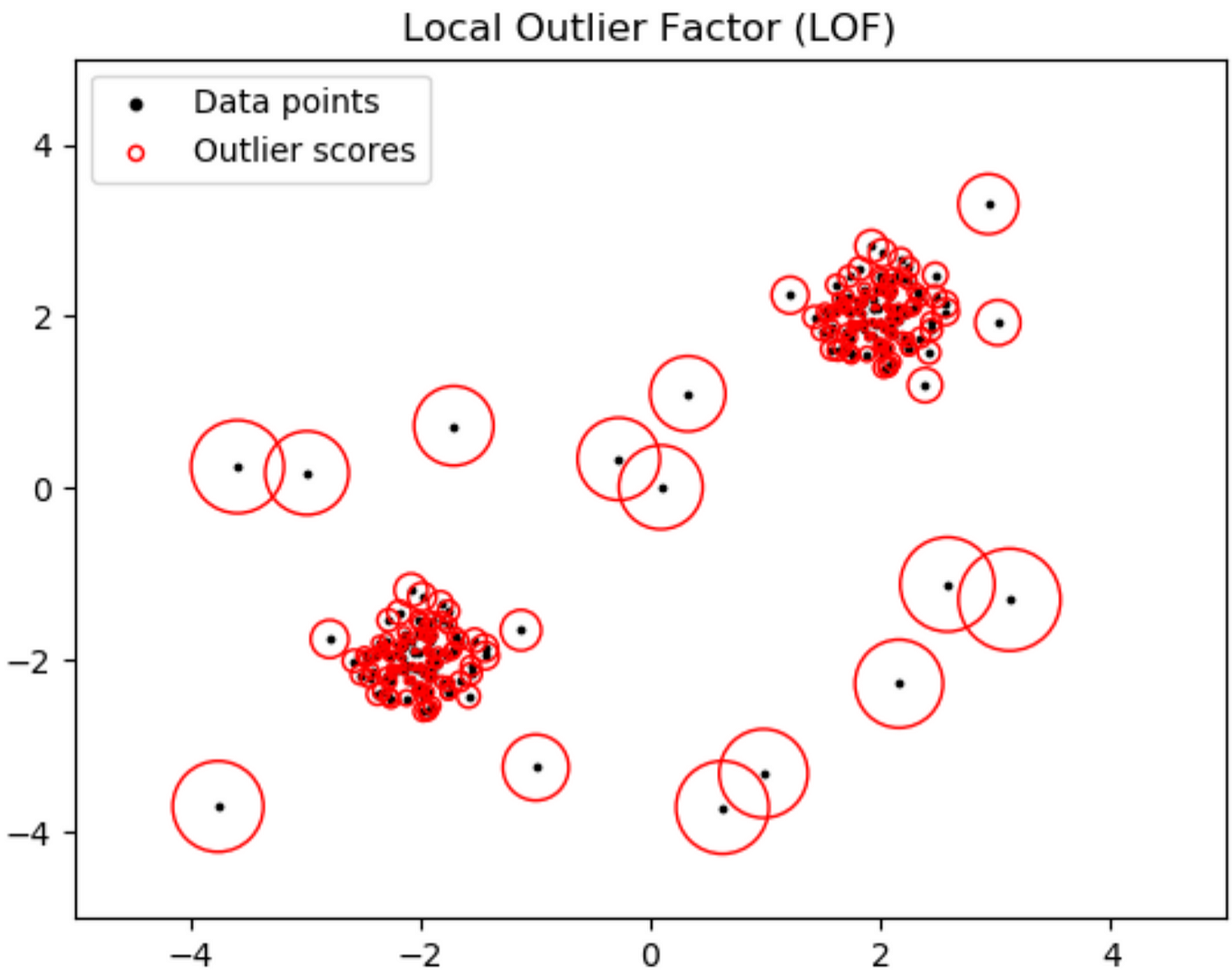
Out[478]:

```
Text(0.5, 1.0, 'Prédiction avec contamination = 0.08')
```



On peut voir que si nous augmentons le taux de contamination, plus le poucentage de notre données serons considéré comme anomalie comme on peut voir sur les graphes précedent.

EXO 3 : Détection des chiffres manuscrits



L'isolation Forest (forêt d'isolation) est un algorithme d'apprentissage non supervisé de Machine Learning qui permet la détection d'anomalies dans un jeu de données. Il isole les données atypiques.

Rappelons que l'apprentissage non supervisé est une des branches du Machine Learning qui possède la particularité de ne pas avoir besoin de données labelisées contrairement à d'autres techniques.

Cet algorithme repose sur le principe que les anomalies sont plus faciles à isoler que les données normales. Il construit un ensemble de forêt, qui partitionnent récursivement le jeu de données en sous-ensembles. Chaque partitionnement est basé sur un attribut choisi au hasard et une valeur seuil tirée entre les valeurs minimale et maximale de cet attribut. Le nombre de partitionnements nécessaires pour isoler une donnée est appelé score d'anomalie. Plus ce score est faible, plus la donnée est considérée comme anormale.

Entrée [479]:

```
digits = load_digits()
Images = digits.images
```

Entrée [480]:

```
X = digits.data
y = digits.target
```

Entrée [481]:

```
print(X.shape)
```

```
(1797, 64)
```

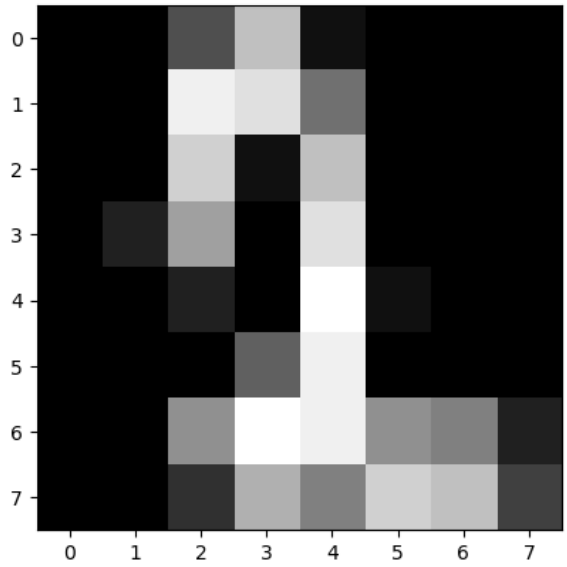
Chaque image dans le jeu de données représente un chiffre manuscrit de 0 à 9. L'image est composée de 8 x 8 pixels, où chaque pixel a une valeur entre 0 et 255 indiquant son niveau de gris. Pour simplifier le traitement, l'image est convertie en un vecteur de taille 64, qui correspond à une ligne de la matrice X. La matrice X contient donc 1797 lignes, une pour chaque image. Le vecteur y contient le label de chaque image, c'est-à-dire le chiffre qu'elle représente.

Entrée [482]:

```
plt.imshow(Images[12],"gray")
```

Out[482]:

```
<matplotlib.image.AxesImage at 0x2225e9fa110>
```



Entrée [483]:

```
model = IsolationForest(random_state = 0,contamination = 0.02)
model.fit(X)
print(model.predict(X))
```

```
[1 1 1 ... 1 1 1]
```

Entrée [484]:

```
np.unique(model.predict(X))
```

Out[484]:

```
array([-1,  1])
```

4. Comme nous utilisons une contamination de 0.02 donc seulement 2% des valeurs de 1797 serons considéré comme anomalies.

si model.predict(X) = 1 donc pas d'anomalie

si model.predict(X) = -1 donc anomalie

Entrée [485]:

```
outliers = model.predict(X)== -1
outliers
```

Out[485]:

```
array([False, False, False, ..., False, False, False])
```

Entrée [486]:

```
unique, counts = np.unique(outliers, return_counts=True)

result = np.column_stack((unique, counts))
print(result)
```

```
[[   0 1761]
 [   1   36]]
```

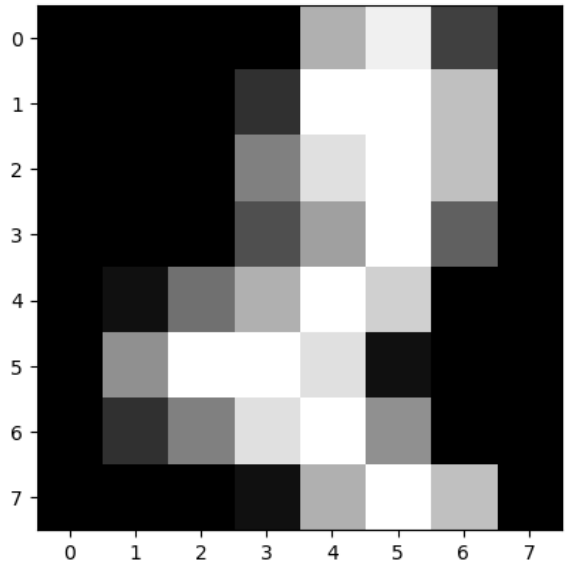
8. si la données presente une anomalie done outliers sera true (1) sinon false (0) comme on peut voir, on a 36 exemples sont considéré comme anomalie.

Entrée [487]:

```
plt.imshow(Images[outliers][0],"gray")
```

Out[487]:

```
<matplotlib.image.AxesImage at 0x2225ea98f10>
```



9. Comme on voit que le premier element de outlier est false donc l'algortithme a definit cet image comme exploitable c'est-à-dire qu'il ne presente pas d'anomalie. De même nous avons très mal à definir au vue d'oeil.

Entrée [488]:

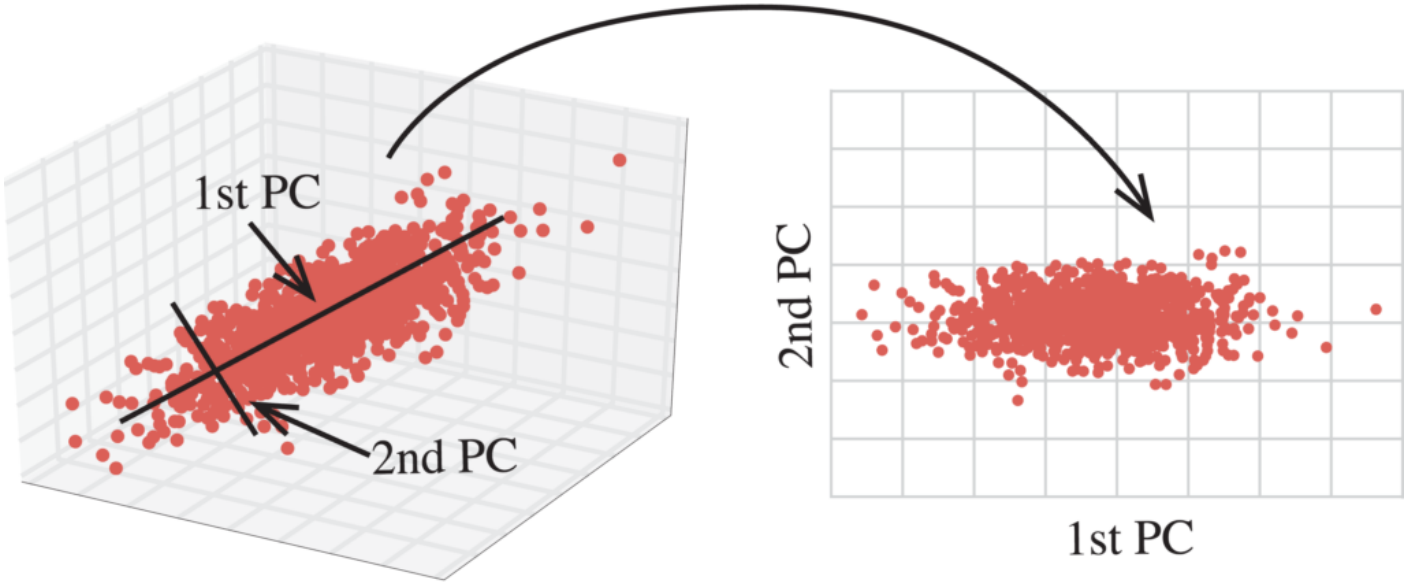
```
model = IsolationForest(random_state = 0,contamination = 0.08)
model.fit(X)
model.predict(X)
outliers = model.predict(X)== -1
unique, counts = np.unique(outliers, return_counts=True)

result = np.column_stack((unique, counts))
print(result)
```

```
[[   0 1653]
 [   1  144]]
```

Si nous changeant l'hyperparmetre condamination, on peut voir que le nombre de données considéré comme presente une anomalie augmente.

EXO 4 : Réduction de la dimensionnalité

La réduction de dimensionnalité fait référence aux techniques qui réduisent le nombre de variables dans un ensemble de données, ou encore projettent des données issues d'un espace de grande dimension dans un espace de plus petite dimension.



A) Téléchargement du jeu de données << digits>>

Entrée [489]:

```
digits = load_digits() # jeu de données
images = digits.images
```

Entrée [490]:

```
X = digits.data # contiennt 1797 images de 8x8 pixels
y = digits.target
```

Entrée [491]:

```
print(X.shape)
print(X.ndim)
print(y.shape)
```

```
(1797, 64)
2
(1797,)
```

X est en 2 Dimention 1797x64

y est en 1 Dimention 1797x1

B) Visualisation des données

Entrée [492]:

```
model = PCA(n_components=2)
X_reduced = model.fit_transform(X)
print(X_reduced.shape)
print(X_reduced.ndim)
```

```
(1797, 2)
2
```

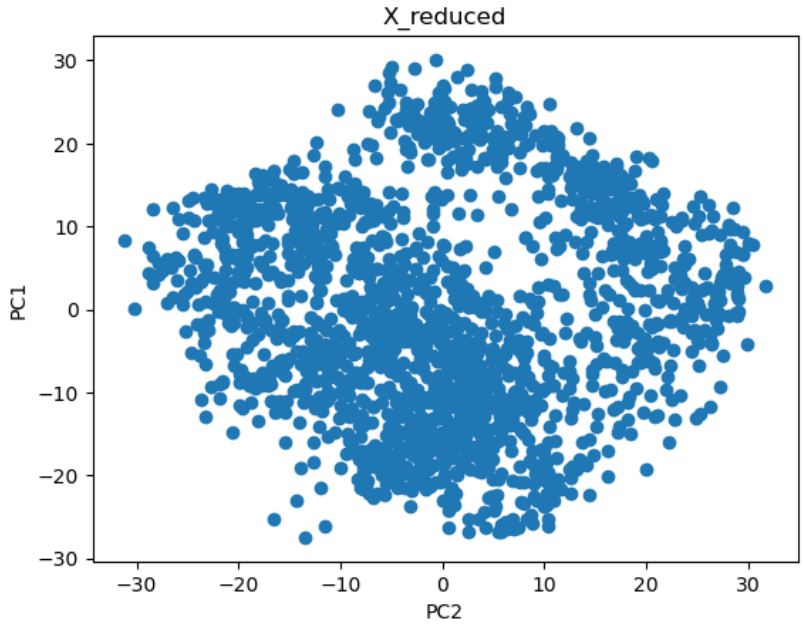
Apres avoir apliqué pca, on passe de 1797x64 à 1797x2

Entrée [493]:

```
plt.scatter(X_reduced[:,0],X_reduced[:,1])
plt.title('X_reduced')
plt.ylabel('PC1')
plt.xlabel('PC2')
```

Out[493]:

```
Text(0.5, 0, 'PC2')
```

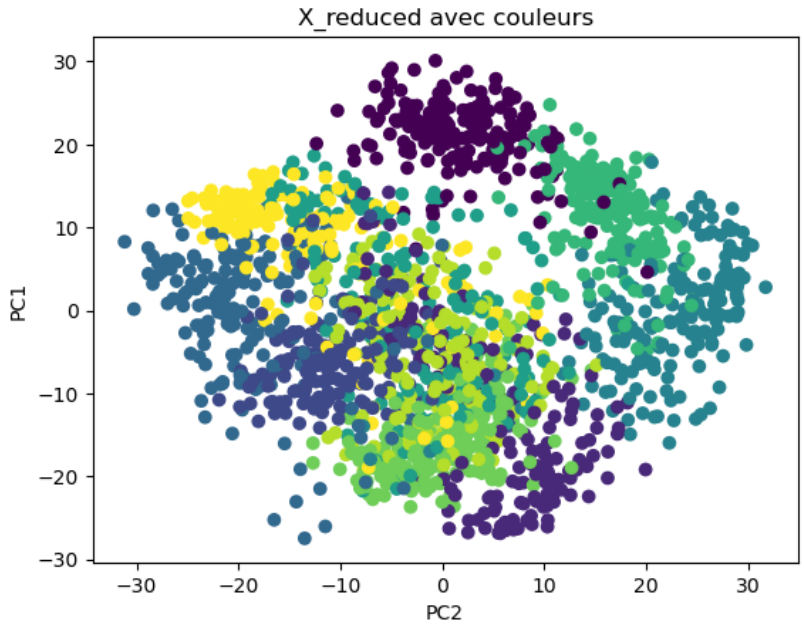


Entrée [494]:

```
plt.scatter(X_reduced[:,0],X_reduced[:,1], c = y)
plt.title('X_reduced avec couleurs')
plt.ylabel('PC1')
plt.xlabel('PC2')
```

Out[494]:

```
Text(0.5, 0, 'PC2')
```



Entrée [495]:

```
y
```

Out[495]:

```
array([0, 1, 2, ..., 8, 9, 8])
```

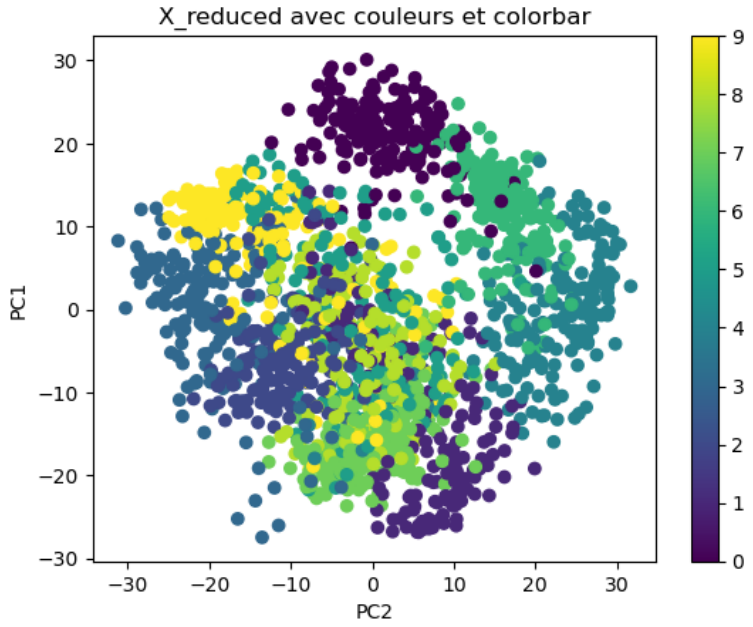
y contient les label des chaque image. c'est a dire le nombre qu'on est sensé visualisé sur chaque image.

Entrée [496]:

```
plt.scatter(X_reduced[:,0],X_reduced[:,1], c = y)
plt.colorbar()
plt.title('X_reduced avec couleurs et colorbar')
plt.ylabel('PC1')
plt.xlabel('PC2')
```

Out[496]:

```
Text(0.5, 0, 'PC2')
```



8. méthode de classification des chiffres basée sur la similarité de leur forme. On voit neuf classes, une pour chaque chiffre de 0 à 9, et chaque classe est représentée par un paquet de données étiqueté. En comparant les paquets de 0 et 9, on constate qu'ils sont proches l'un de l'autre, car les chiffres 0 et 9 ont une forme très semblable. En revanche, les paquets de 0 et 1 sont éloignés, car les chiffres 0 et 1 ont une forme très différente. Cette méthode permet de regrouper les chiffres qui se ressemblent et de les distinguer de ceux qui ne se ressemblent pas.

9. L'analyse en composantes principales (ACP) est une technique qui permet de réduire la dimensionnalité d'un ensemble de données. Elle consiste à trouver les axes principaux qui expliquent le mieux la variabilité des données. La première composante principale (PC1) est l'axe qui maximise la variance des données projetées sur cet axe. La deuxième composante principale (PC2) est l'axe orthogonal à PC1 qui maximise la variance restante

Entrée [497]:

```
model.components_.shape
```

Out[497]:

```
(2, 64)
```

9. Le model contiennt 64 variables à 2 dimentions (PC1 et PC2)

C) Compression de données

Entrée [498]:

```
d = 64 # dimention de X
```

Entrée [499]:

```
model = PCA(n_components=d)
X_reduced = model.fit_transform(X)
model.explained_variance_ratio_
```

Out[499]:

```
array([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])
```

Entrée [500]:

```
np.cumsum(model.explained_variance_ratio_) #somme cumulative des éléments d'un tableau.
```

Out[500]:

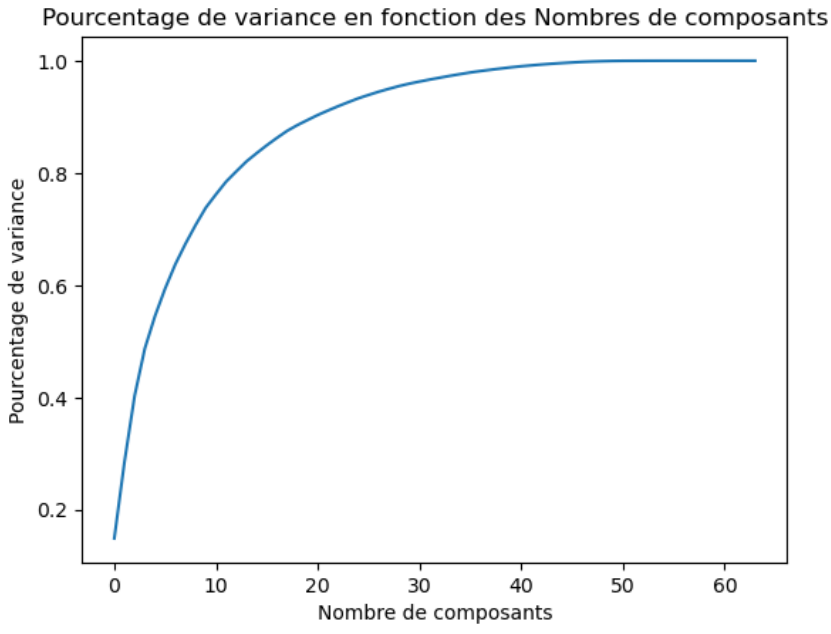
```
array([0.14890594, 0.28509365, 0.40303959, 0.48713938, 0.54496353,
       0.59413263, 0.6372925 , 0.67390623, 0.70743871, 0.73822677,
       0.76195018, 0.78467714, 0.80289578, 0.82063433, 0.83530534,
       0.84940249, 0.86258838, 0.87506976, 0.88524694, 0.89430312,
       0.9031985 , 0.91116973, 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.        ])
```

Entrée [501]:

```
plt.plot(np.cumsum(model.explained_variance_ratio_))
plt.title('Pourcentage de variance en fonction des Nombres de composants')
plt.ylabel('Pourcentage de variance')
plt.xlabel('Nombre de composants')
```

Out[501]:

```
Text(0.5, 0, 'Nombre de composants')
```



6. D'apres le graphique on peut voir qu'on atteint 90% de variance à partir de 20 composants

Entrée [502]:

```
np.argmax(np.cumsum(model.explained_variance_ratio_)>0.99) #index de l'élément maximum dans les tableaux
```

Out[502]:

```
40
```

7. On atteint les 99% de varaince à partir de 40 composants.

Les composants sont des combinaisons linéaires des variables originales qui capturent le maximum de la variance des données. Plusieurs composants signifient qu'il faut plus d'un composant pour représenter fidèlement les données. Le nombre optimal de composants dépend du critère choisi comme ici le pourcentage de variance.

Entrée [503]:

```
model = PCA(n_components=2)
X_reduced = model.fit_transform(X)
```

Entrée [504]:

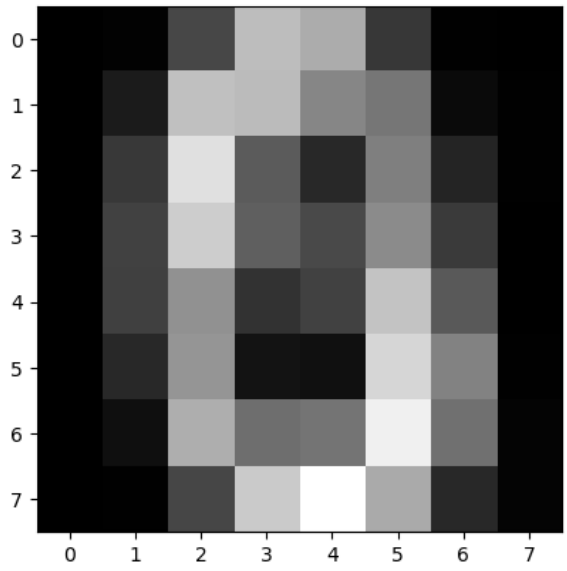
```
X_recovered = model.inverse_transform(X_reduced)
```

Entrée [505]:

```
plt.imshow(X_recovered[0].reshape((8,8)),'gray')
```

Out[505]:

```
<matplotlib.image.AxesImage at 0x2225ed65650>
```

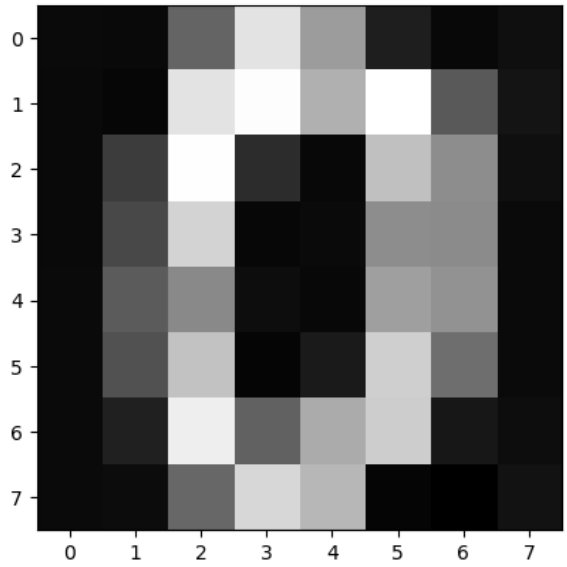


Entrée [506]:

```
model = PCA(n_components=40)
X_reduced = model.fit_transform(X)
X_recovered = model.inverse_transform(X_reduced)
plt.imshow(X_recovered[0].reshape((8,8)),'gray')
```

Out[506]:

```
<matplotlib.image.AxesImage at 0x2225eeeb610>
```



10. Une analyse de la relation entre la n_components et la qualité d'image montre que plus la n_components est élevée, plus l'image est nette et détaillée. En revanche, si la n_components est faible, l'image est floue et perd de sa précision. Cela s'explique par le fait que la réduction de dimensionnalité implique une perte d'information et une simplification des caractéristiques de l'image.

Entrée [507]:

```
scaler = MinMaxScaler()
data_rescaled = scaler.fit_transform(X)
pca = PCA(n_components=0.40)
pca.fit(data_rescaled)
reduced = pca.transform(data_rescaled)
```

Entrée [508]:

```
pca.n_components_
```

Out[508]:

```
3
```

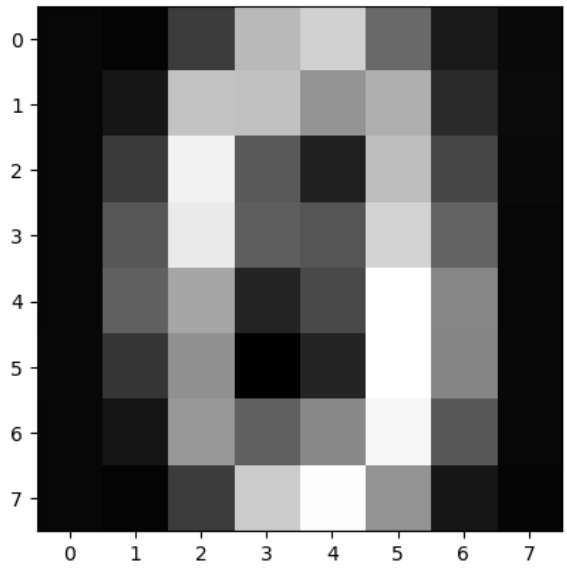
Nombre de composant utilisé est 3

Entrée [509]:

```
X_recouv = pca.inverse_transform(reduced)
plt.imshow(X_recouv[0].reshape((8,8)),'gray')
```

Out[509]:

```
<matplotlib.image.AxesImage at 0x2225ef5a110>
```



On peut voir qu'apres la mise en échelle on peut voir une dégradation de qualité d'image.

Entrée [510]:

```
reduced
```

Out[510]:

```
array([[ 0.06113739,  1.37811679, -0.53875543],
       [ 0.37573724, -1.35466438,  0.2269637 ],
       [ 0.37052705, -0.67597542,  0.16865822],
       ...,
       [ 0.61313384, -0.52192977,  0.35380417],
       [-0.23032906,  0.82527455, -0.59848032],
       [-0.02290219,  0.34818667,  0.70369523]])
```

12. si la n_components est faible, l'image est floue et perd de sa précision. Cela s'explique par le fait que la réduction de dimensionnalité implique une perte d'information et une simplification des caractéristiques de l'image.