TP3 – Analyse en composantes principales

4A : INGENIERIE DES DONNÉES

Analyse de la base de données

Nous utilisons un dataset compris dans la bibliothèque *sklearn*, nommé *"digits"*, qui contient des images de caractères manuscrits, représentées en matrices de niveaux de gris allant de 0 à 16.

Chargement et analyse du dataset

Après avoir chargé le dataset, nous pouvons obtenir des informations sur avec les commandes suivantes :

```
print(digits["feature_names"])
```

```
['pixel_0_0', 'pixel_0_1', 'pixel_0_2', ..., 'pixel_7_5', 'pixel_7_6', 'pixel_7_7']
```

```
print(digits["target_names"])
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
print(digits["data"].shape)
```

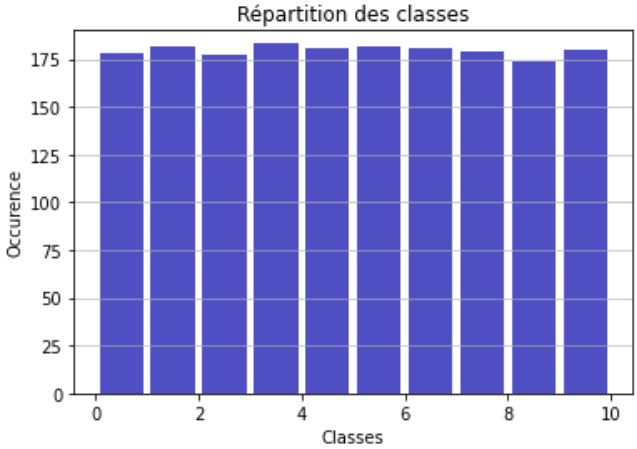
```
(1797, 64)
```

Nous apprenons ici que les descripteurs sont les niveaux de gris des pixels des images, nommés en lignes et colonnes, et qu'ils sont au nombre de **64**, représentant des images carrées de 8 par 8 pixels. Nous apprenons également que les classes sont les chiffres allant de **0 à 9**. Enfin, la dernière information tirée est le nombre d'exemples : **1797** images constituent ce dataset.

Nous pouvons obtenir le nombre d'exemples par classe pour en étudier leur répartition :

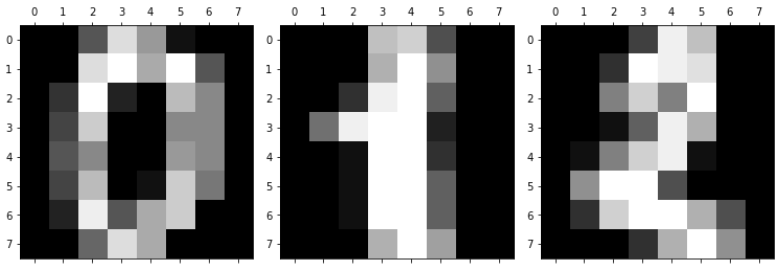
```
unique, counts = np.unique(digits["target"], return_counts=True)
print(dict(zip(unique, counts)))
```

```
{0: 178, 1: 182, 2: 177, 3: 183, 4: 181, 5: 182, 6:
181, 7: 179, 8: 174, 9: 180}
```



On peut voir que les classes sont équitablement réparties.

Aperçu des images :



Entraînement avec le KNN

Nous répétons l'entraînement plusieurs fois, pour comparer les résultats, et effectuons à chaque fois les mêmes opérations :

- Nous séparons la base initiale de manière aléatoire en 2 bases :

```
randstate = rd.randint(0,2147483647)
X_train, X_test, y_train, y_test = train_test_split(digits["data"], digits["target"], random_state=randstate, train_size=0.7)
```

- Nous entraînons ensuite notre modèle avec la base d'entrainement :

```
OneNN = KNeighborsClassifier(n_neighbors=1)
OneNN.fit(X_train, y_train)
```

- Nous testons notre modèle sur la base de test :

```
KNN_predict.append(OneNN.predict(X_test))
```

- Nous étudions le score de reconnaissance du modèle avec la fonction *accuracy_score()* :

```
accuracy_score(y_test, KNN_predict[i]))
```

Les résultats obtenus sont les suivants, pour 10 itérations :

```
Random state : 1644660476 | Taux de reconnaissance
0.9907407407407407

Random state : 2109268137 | Taux de reconnaissance
0.9944444444444445

Random state : 361466871 | Taux de reconnaissance
0.987037037037037

Random state : 1096202591 | Taux de reconnaissance
0.9888888888888889

Random state : 428107814 | Taux de reconnaissance
0.9833333333333333

Random state : 1212111529 | Taux de reconnaissance
0.9851851851851852

Random state : 2090422850 | Taux de reconnaissance
0.9851851851851852

Random state : 636991712 | Taux de reconnaissance
0.9888888888888889

Random state : 164594853 | Taux de reconnaissance
0.9925925925925926

Random state : 268844137 | Taux de reconnaissance
0.9851851851851852
```

On peut voir que la méthode du KNN est stable et donne de bons résultats (**≈98%** de précision).

Analyse en composantes principales

Solveur PCA

Nous entraînons ici le solveur avec 64 composantes principales :

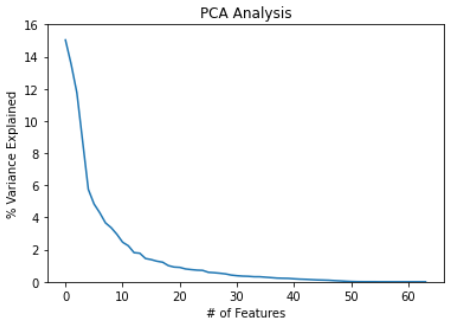
```
from sklearn.decomposition import PCA

X_train_64 = X_train
pca = PCA(n_components = 64)
pca.fit(X_train_64)
```

On utilise une base d'entraînement distincte pour entraîner le modèle. La base de test sert à confirmer l'entraînement avec des éléments que le modèle n'a jamais rencontrés, et donc sur lequel il n'a pas pu s'entraîner, afin de ne pas biaiser les résultats.

Inertie expliquée

Nous pouvons extraire les valeurs propres pour tracer le graphe de l'inertie expliquée ci-contre :

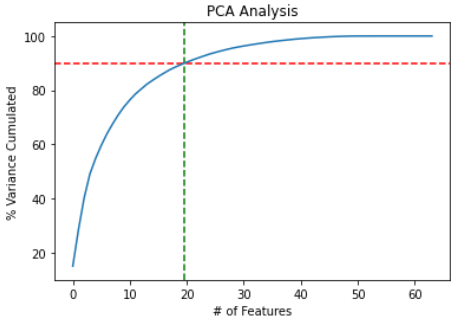



```
exp_var = pca.explained_variance_ratio_*100
```

Ayant ici un problème à 64 dimensions, nous ne pouvons pas appliquer le critère de Catell pour déterminer le nombre de composantes à conserver, celui-ci étant optimal pour des problèmes à faibles dimensions.

Inertie cumulée

Nous pouvons calculer l'inertie cumulée à partir des valeurs propres extraites plus haut et ainsi obtenir le graphe ci-contre :




```
cum_var = np.cumsum(np.round(variance, decimals=3))
```

En appliquant le critère de Joliffe à **90%**, nous voyons qu'un nombre optimal de composantes principales à conserver serait **20**. Nous allons utiliser ce nombre pour les exemples suivants.

Classification

M premières composantes principales

Comme dans le cas précédent, nous allons créer notre modèle et adapter notre dataset en fonction du PCA et du nombre de composantes principales retenues :

```
pca = PCA(n_components=20)
X_train_20 = pca.fit_transform(X_train)
X_test_20 = pca.transform(X_test)
```

Nous décidons ici d'utiliser 20 composantes (résultat découlant du critère de **Joliffe**). La fonction *fit_transform* permet de réduire le nombre de dimension et entraîner correctement le modèle. La fonction *transform* permet de réduire le nombre de dimensions.

La fonction *shape* nous permet de vérifier que la transformation s'est bien effectuée :

```
X_train_20.shape
```

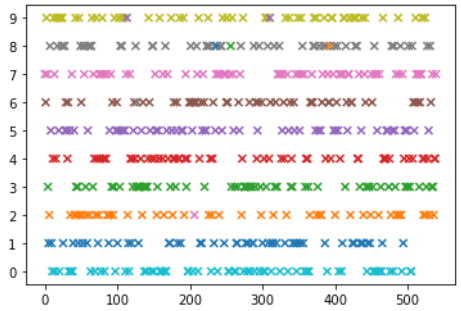
```
(1257, 20)
```

Nous entraînons ensuite notre KNN avec ce dataset redimensionné et testons sa précision avec la fonction *accuracy_score()* :

```
OneNN.fit(X_train_20, y_train)
KNN_predict = OneNN.predict(X_test_20)
```

```
Taux de reconnaissance 0.987037037037037
```

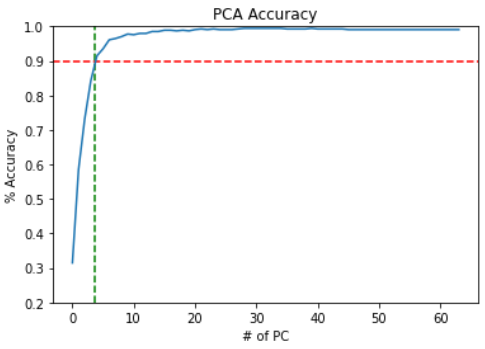
Nous utilisons les mêmes méthodes qu'auparavant pour représenter les classes, et obtenons le graphe ci-contre :



Extension de l'analyse

Nous réitérons toute la méthode de classification que nous venons de voir, mais en faisant cette fois varier le nombre de composantes principales de **1 à 64**, pour visualiser l'évolution du taux de reconnaissance en fonction du nombre de composantes retenues.

```
for i in range(digits["data"].shape[1]):
    pca = PCA(n_components=i+1)
    X_train_i = pca.fit_transform(X_train)
    X_test_i = pca.transform(X_test)
```



Le graphique ci-dessus révèle qu'à partir de **4 composantes principales**, nous atteignons **90%** de précision. Nous pouvons donc réduire la dimension de ce dataset de 64 à 4 ou 5 sans perdre en performance et en précision.

Compression

La réduction de dimension constitue la phase de compression, car des données disparaissent dans le processus et potentiellement perdues. La reconstruction va nous permettre de mesurer cette perte.

Etude de la reconstruction

Récupération des valeurs propres et vecteurs propres

Comme auparavant, nous réduisons le nombre de dimensions de notre dataset :

```
pca = PCA(n_components=20)
X_train_20 = pca.fit_transform(X_train)
```

Nous pouvons en extraire les valeurs propres et vecteurs propres avec les fonctions *explained_variance_* et *components_* :

```
eigenvalues = pca.explained_variance_
eigenvectors = pca.components_
```

A l'aide de la fonction *shape*, nous vérifions la bonne extraction des valeurs et vecteurs propres :

```
print("Eigenvalues :", eigenvalues.shape)
print("Eigenvectors :", eigenvectors.shape)
```

```
Eigenvalues : (20,)
Eigenvectors : (20, 64)
```

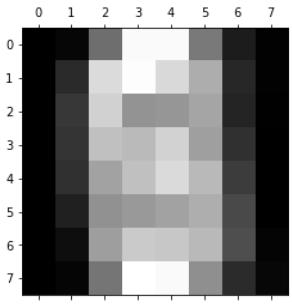
Ce qui correspond aux dimensions attendues.

Image moyenne

Dans le cadre de la reconstruction, nous avons besoin d'obtenir une image moyenne :

```
digits_mean = []
for i in range(digits["data"].shape[1]):
    digits_mean.append(mean(digits["data"][:,i]))
```

Nous obtenons l'image moyenne suivante :



Vecteurs de compression

Dans l'opération de décompression, il nous est nécessaire d'avoir des vecteurs de compression/décompression. Ces vecteurs sont propres à chaque image et sont exprimés sous la forme :

$$C = P_M^T(X - m) = [c_1 \quad \dots \quad c_M]^T$$

Avec P_M^T la matrice de vecteurs propres, X l'image traitée et m l'image moyenne.

```
C = [] #matrice des vecteurs de compression/decompression
X_m = [] #matrice des X - m

for i in range(digits["data"].shape[0]):
    X_m.append(digits.data[i] - digits_mean) #calcul de la différence X - m
    c_i = [] #vecteur c_i actuel
    for j in range(20):
        c_i.append(np.dot(eigenvectors[j], X_m[i])) #produit scalaire des 2 matrices
    C.append(c_i)
```

Cette boucle nous permet d'obtenir les vecteurs de compression/décompression, de dimensions (1797, 20).

Reconstruction

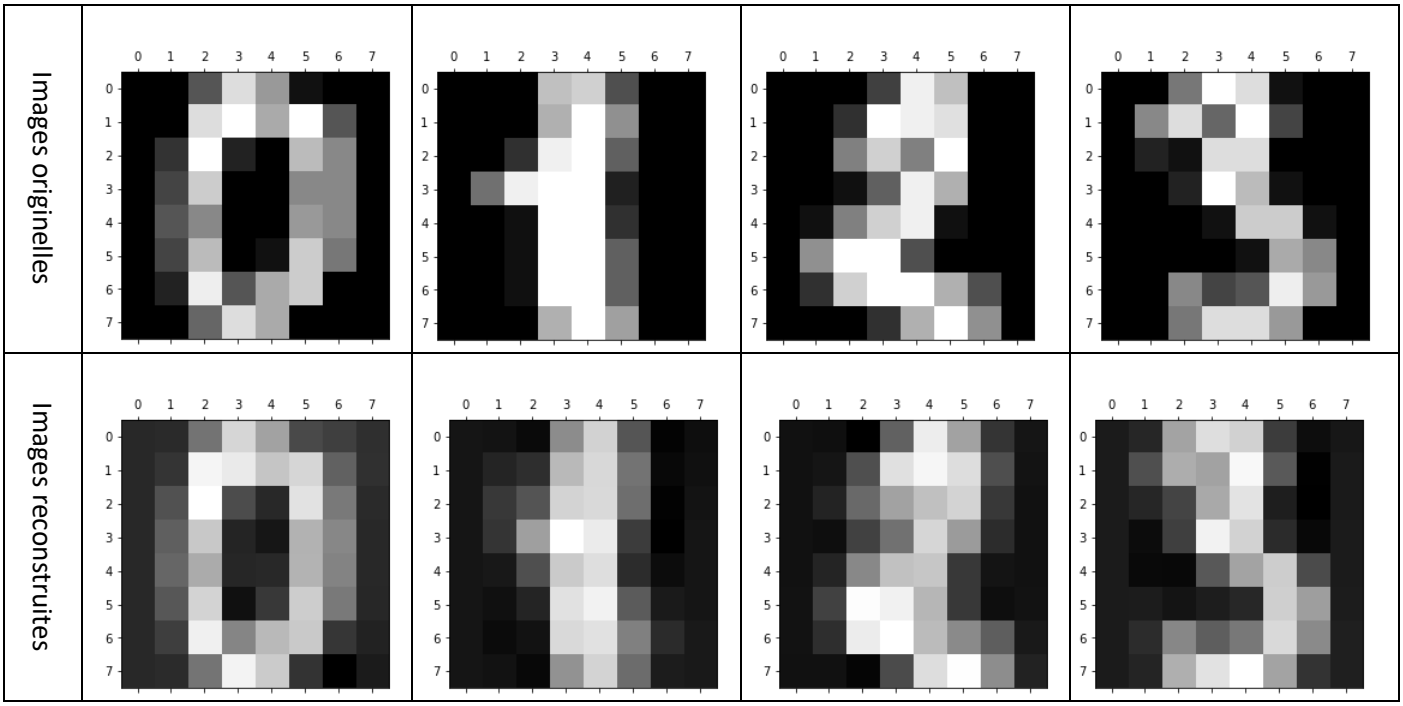
La reconstruction se fait suivant cette formule :

$$X \approx m + P_M C = m + \sum_{i=0}^{M} c_i e_i$$

Nous appliquons cette formule :

```
R = []
for i in range(digits["data"].shape[0]):
    R.append(digits_mean) #ajout de l'image moyenne
    for j in range(20):
        R[i] += C[i][j]*eigenvectors[j] #produit des vecteurs de compression/decompression et des vecteurs propres
```

En affichant les images reconstruites et d'origines, nous pouvons comparer l'efficacité de la compression/décompression quant à la perte d'information :



Les résultats semblent assez corrects et nous permettent de reconnaître les caractères affichés, en tant qu'humain.

Erreur de reconstruction

Locale

Nous pouvons obtenir l'erreur de reconstruction d'une image en calculant pour chaque pixel le carré de la différence entre l'image d'origine et l'image reconstruite, puis de faire la moyenne des pixels :

```
R_error = (R[0] - digits.data[0])**2
R_error_mean = mean(R_error)/16
print(R_error_mean)
```

```
0.06341121125741067
```

La division par 16 correspond à une normalisation, les valeurs des niveaux de gris allant de **0 à 16**.

Nous voyons que l'erreur de reconstruction est ici d'environ **6.3%**, ce qui est très faible compte-tenu du nombre de dimensions que nous avons supprimées.

Globale

Nous pouvons appliquer cette même formule pour chaque image, puis calculer la moyenne de toutes les images et obtenir une erreur globale de reconstruction :

```
R_error = []
R_error_mean = []
```

```
for i in range(digits["data"].shape[0]):
    R_error.append((R[i] - digits.data[i])**2)

for i in range(digits["data"].shape[0]):
    R_error_mean.append(mean(R_error[i]))
R_error_mean = np.array(R_error_mean)
R_error_mean /= 16

R_error_global = mean(R_error_mean)
print(R_error_global)
```

```
0.12485873495555112
```

Nous obtenons donc une erreur de reconstruction globale d'environ **12.5%**, ce qui est déjà plus conséquent.

Extension de l'analyse

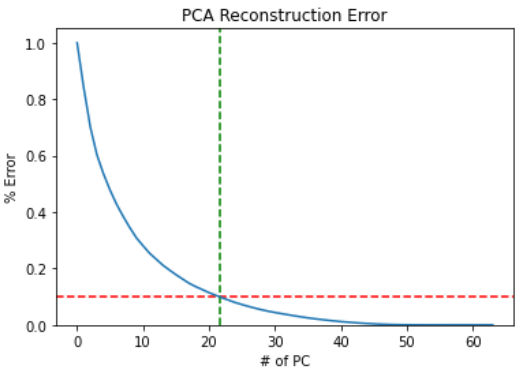
Afin de visualiser l'évolution de l'erreur de reconstruction en fonction du nombre de composantes retenues, nous réitérons toute la méthode de calcul d'erreur que nous venons de voir, mais en faisant cette fois varier ne nombre de composantes principales de 1 à 64.

```
reconstruct_error_global = []
for i in range(digits["data"].shape[1]):
    #----------------PCA----------------#
    ...
    #-------------COMPRESSION-------------#
    ...
    #-----------RECONSTRUCTION-----------#
    ...
    #---------------ERREUR---------------#
    ...

    print("Erreur de reconstruction globale pour N =",i+1, ":", reconstruct_error_global[i])
```

```
Erreur de reconstruction globale pour N = 1 : 0.999138870272826
Erreur de reconstruction globale pour N = 2 : 0.8404664119585734
Erreur de reconstruction globale pour N = 3 : 0.7012010581046921
...
Erreur de reconstruction globale pour N = 62 : 3.1327520689413805e-30
Erreur de reconstruction globale pour N = 63 : 3.1327702316496346e-30
Erreur de reconstruction globale pour N = 64 : 3.1327420786502367e-30
```

Nous obtenons le graphe ci-contre :



Nous pouvons constater que l'erreur est maximale lorsqu'une seule composante n'est retenue, elle approche en effet les **100%**.

Pour passer la barre des **10%** d'erreur, il nous faut conserver au moins **22** composantes principales.