Le clustering

1 Initiation à Python et à l'usage des principales librairies

L'objectif de cette première partie de projet est d'utiliser des modules de Python afin de réaliser à la fois les méthodes de classification hiérarchique ascendante et de tester la méthode des centroïdes (*k-means*), ceci afin de pouvoir rassembler automatiquement les données en différentes classes.

Python est un langage de programmation de haut niveau. C'est un langage très versatile qui permet à la fois d'utiliser la programmation orienté-objet et de traiter les données à l'aide des librairies déjà existantes.

Cette première partie de projet va consister à utiliser la puissance de cet environnement pour réaliser avec un minimum de programmation des méthodes d'analyse de données, plus précisément de Clustering. Afin de découvrir Python et d'avoir des points d'entrée de références, vous sont fournis différents textes et livres, certains brefs, d'autres livres de référence, mais tous libres de droits. Vous allez appliquer les méthodes de Clustering sur trois jeux de données.

Quelques commandes de Python utiles:

- Si vous souhaitez une aide sur un objet object : `help(object)`
- Charger le module `module` avec un nom symbolique abrégé `md` : `import module as md`
- La liste des variables en mémoire locales `locals()`, globales `globals()`
- Effacer la variable variable : `del variable`
- Les fonctions et variables associés à un objet `X` : `dir(X)`

Quelques commandes du module `numpy` qui est une librairie mathématique (notamment de manipulation des matrices). Un tableau [1, 2, 3] peut être crée sous format matrice ou liste :

```
1 import numpy as np
2 # Creation d une liste
3 X = [1 , 2 , 3]
4 # Creation d une matrice
5 X = np.array([1, 2, 3])
```

Quels sont les avantages à utiliser une structure `array` :

- on peut créer un tableau multidimensionnel `X = np.array([(1, 2, 3),(4, 5, 6)])` ;
- les éléments dans la structure sont homogènes (forcément du même type) ;
- les opération point à point sont prévues ;
- les structures `Numpy array` sont associées à de nombreuses fonction.

Quelques fonctions élémentaires Numpy :

- Création d'un vecteur X = [1, 1, 1, 1, . . .] de taille 1 × n: `X = np.ones(n)`

- Opérations arithmétiques élément par élément de deux arrays `X` et `Y` de même longueur :
  - Addition : `np.add(X,Y)`
  - Multiplication : `np.multiply(X,Y)`
  - Division : `np.divide(X,Y)`
- Concaténation:
  - de deux arrays `X` et `Y` numériques avec l'axe axis sur lequel fusionner: `data=np.concatenate((X, X), axis=0)`
  - de deux strings `X` et `Y` : `X+Y`

1.1 Tests élémentaires sous Python

1.1.1 Création du corpus de test

Tout d'abord fin de tester les programmes de *clustering* proposés dans les librairies standards, vous allez créer un nuage de points correspondant à l'acquisition de 2 classes d'individus chacun associé à une distribution Gaussienne bidimensionnelle pour un vecteur $\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2$.

La première classe est composée de 128 individus et est associée à la fonction de densité de la loi normale $\mathcal{N}_2((2,2)^T, 2I_2)$:

$$f(\mathbf{x}) = f(x_1, x_2) = \frac{1}{4\pi} \exp\left\{-\frac{1}{2}\left(\mathbf{x} - \begin{bmatrix} 2 \\ 2 \end{bmatrix}\right)^t \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \left(\mathbf{x} - \begin{bmatrix} 2 \\ 2 \end{bmatrix}\right)\right\}$$

La seconde classe est composée de 128 individus et est associée à la fonction de densité de la loi normale $\mathcal{N}_2((-4,-4)^T, 6I_2)$:

$$f(\mathbf{x}) = f(x_1, x_2) = \frac{1}{4\pi} \exp\left\{-\frac{1}{2}\left(\mathbf{x} - \begin{bmatrix} -4 \\ -4 \end{bmatrix}\right)^t \begin{bmatrix} 6 & 0 \\ 0 & 6 \end{bmatrix} \left(\mathbf{x} - \begin{bmatrix} -4 \\ -4 \end{bmatrix}\right)\right\}$$

Pour créer ces données de test, vous pouvez utiliser Python comme suit:

```
import numpy as np
import matplotlib.pyplot as plt
import random
import os

# Simulation de n=256 donnees X=(X1,X2) suivant une loi normale
    bidimensionnelle de moyenne (0,0), de la variance Var(X1)=1, Var(X2)=1
     et la covariance Cov(X1,X2)=0.5
mean = np.array([0, 0])
cov = np.array([[1, 0.5], [0.5, 1]])
X = (np.random.multivariate_normal(mean, cov, 256))

# Affichage des donnees
plt.plot(X[:,0],X[:,1],"o",label = 'Individu')
plt.legend
```

Q. 1 Adapter les lignes de code décrit ci-dessus pour construire votre ensemble de test de taille 256 individus par 2 mesures correspondant aux deux classes. Afficher vos deux classes avec des symboles/couleurs différents selon la classe.

1.2 Méthode de K-means

1.2.1 Test de la Méthode de K-means

Q. 1 Effectuer le clustering de vos données data en K clusters par la méthode de K-means à l'aide de l'objet `KMeans` *du module* `cluster` *de la bibliothéque* `sklearn`*:*

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=K, n_init=1, init='k-means++')
kmeans.fit(data)
```

Q. 2 Essayez différentes valeurs de nombre de clusters `K`*. Interpréter le résultat de clustering donné par l'attribut* `kmeans.labels_`

Q. 3 Afficher graphiquement le résultat du clustering avec deux couleurs/symboles à l'aide de `kmeans.labels_`*. Etudier les différences avec la "vraie" classification.* `adjusted_rand_score` *du module* `metrics` *de la bibliothéque* `sklearn` *permet de calculer l'erreur entre la classification obtenue et la classification "vraie". Etudier l'aide pour comprendre la mesure.*

Q. 4 Utiliser la fonction `adjusted_rand_score` *pour calculer le taux d'erreur.*

Q. 5 A quoi sert le paramètre `n_init` *? Essayer une autre valeur que 1 et commenter les différences (ou l'absence de différences).*

1.2.2 Choix du "bon" nombre de clusters

Q. 1 Séparer les données `data` *en* $K = 2, 3, 4, 5, 6$ *clusters à l'aide de la fonction* `KMeans`

Q. 2 Calculer les largeurs de silhouettes des observations et le coefficient de silhouette :

```
from sklearn.metrics import silhouette_samples, silhouette_score

K = 3
kmeans = KMeans(n_clusters=K, n_init=1, init='k-means++').fit(data)
cluster_labels=kmeans.labels
silhouette_avg = silhouette_score(data, cluster_labels)

print("Pour le nombre de clusters K = {}, le coefficient de
    silhouette est {}".format(K, silhouette_avg))
# Calcul des largeurs de silhouette :
sample_silhouette_values = silhouette_samples(data, cluster_labels)
```

Q. 3 Extraire le coefficient d'inertie selon le nombre de clusters (champ `kmeans.inertia_`*). Afficher le coefficient de silhouette et l'inertie selon le nombre de classes.*

Q. 4 Choisir le meilleur paramètre `K` *pour le clustering.*

1.3 Clustering ascendant hiérarchique

Q. 1 Appliquer l'algorithme de clustering ascendant (CAH) du module cluster de la bibliothèque `scipy` *en utilisant l'objet* `hierarchy` *:*

```
from scipy.cluster.hierarchy import linkage as CAH
# Calculer les dissimilarites entres les clusters a chaque etape:
Z_complete = CAH(data, method='complete', metric='euclidean')
```

Q. 2 Afficher le dendrogramme sans seuil :

```
from scipy.cluster.hierarchy import dendrogram
# Calculer les dissimilarites entres les clusters a chaque etape:
plt.title("CAH")
d = dendrogram(Z_complete, color_threshold=0)
```

Q. 3 Afficher le dendrogramme pour $K = 3$ clusters. Pour cela, choisir le seuil adapté (paramètre `color_threshold`*). Vous pouvez extraire la classification en $K = 3$ clusters en utilisant les commandes suivantes (en choisissant le seuil) :*

```
from scipy.cluster.hierarchy import fcluster
groupes_cah = fcluster(Z, t=seuil, criterion='distance')
# Ajouter la ligne horizontale de la coupe
plt.axhline(y=seuil, c='grey', lw=1, linestyle='dashed')
```

Q. 4 Choisir le seuil automatiquement à l'aide de la troisième variable `Z_complete[,2]` *de* `Z_complete` *qui donne la hauteur de dendrogramme des clusters fusionnés à chaque étape.*

Q. 5 Afficher graphiquement le résultat du clustering avec trois couleurs/symboles à l'aide de `groupes_cah`*. Analyser la classification.*

Q. 6 Comparer les résultats de clustering en $K = 2$ clusters avec les quatre méthodes : lien simple `single`*, lien complet* `complete`*, lien moyen* `average`*, et la dissimilarité de Ward* `ward`*. Présenter quatre dendrogrammes sur le même graphe. Pour calculer l'efficacité, vous pouvez utiliser* `adjusted_rand_score`*et ainsi comparer quantitativement les techniques. Comparer avec le résultats obtenus avec K-Means.*

2 Application dans un contexte de données réelles

2.1 Utilisation de la méthode K-Means pour la réduction de couleur

Nous voulons utiliser la méthode K-means afin de réduire le nombre de couleurs d'une image codée à l'origine sur 16 millions de couleurs. Pour cela nous devons utiliser une classification en K classes où K correspondra aux nombres de couleurs que l'on conserve après réduction. Les nouvelles couleurs seront les centres des classes (les couleurs moyennes résumant au mieux l'information présente dans l'image).

Q. 1 Effectuer la procédure suivante :

- *Charger l'image* `visage.bmp`

```
import matplotlib.image as mpimg

img=np.float32(mpimg.imread('visage.bmp'))
```

- *Organiser les données de l'image couleur sous la forme d'un tableau 256×256 lignes et 3 colonnes : nous considérons chaque pixel comme un individu.*

  Ceci peut se faire à travers la commande suivante np.reshape().

- *Appliquer l'algorithme K-Means sur la tableau de données de 256×256 lignes et 3 colonnes avec K le nombre de couleurs que vous souhaitez conserver après transformation.*

  Nous allons maintenant pouvoir appliquer l'étape de codage qui consiste à remplacer chaque couleur de l'image d'origine par son code (le numéro de la classe du pixel). L'étape de décodage consiste à remplacer le code dans l'image compressée par la couleur associée, à savoir le centre de la classe affectée au pixel

  `kmeans.cluster_centers_[kmeans.labels_[ind],:]`*.*

- *Après le calcul du clustering, remplacer les pixels couleurs de l'image d'origine par la valeur (R,G,B) du centre de la classe affectée au pixel (étape de codage-décodage). Vous obtenez alors un nouveau tableau de* 256×256 *lignes et 3 colonnes qui ne contient que* K *valeurs de couleur différentes.*
- *Ré-organiser les données de l'image couleur sous la forme d'un tableau* $256 \times 256 \times 3$*. Visualiser l'image résultant de la compression avec* `plt.imshow(res/255)`

Q. 2 Commenter et faire plusieurs tests en faisant varier K*.*

2.2 Clustering de données de températures

Cet exercice retranscrit une démarche de classification automatique d'un ensemble de mesures sur différentes villes (35 observations) décrits par leurs valeurs de températures moyennes par mois, sur l'année et la variation. De plus, le tableau contient la valeur de longitude, latitude et la région (ex. 'ouest'). L'objectif est d'identifier des groupes de villes, partageant des caractéristiques similaires. Nous utiliserons les approches de clustering hérarchiques et la méthode des centres mobiles (k-means). Les données sont dans le fichier `temperatures.csv`*. Chargez les données grâce à la librairie* `pandas`*:*

```
import pandas as pd

data_temperature = pd.read_csv("temperatures.csv",sep=";",decimal="
    .",header=0,index_col = 0)
n=len(data_temperature)
```

Le paramètre `header = 0` *indique que la première ligne contient le nom des variables et* `index_col = 0` *indique que la première colonne contient le nom des individus (observations).*

Ensuite dans le tableau, toutes les colonnes ne vont pas être utilisées pour établir les clusters, à savoir 'Region', 'Moyenne', 'Amplitude', 'Latitude', 'Longitude'. Nous allons donc les enlever avec l'instruction :

```
data=data_temperature.drop(columns=['Region','Moyenne','Amplitude',
    'Latitude','Longitude']).
```

2.2.1 Classification hiérarchique ascendante

Q. 1 Calculer la matrice de dissimilarité entre tous les individus. Choisir une mesure de dissimilarité entre classe et construire l'arbre. Si l'on a besoin de normaliser les données, on peut utiliser la commande `scale`*.*

Q. 2 À l'aide du dendrogramme, choisir un nombre de classe et donc un seuil. Appliquer ce seuil.

Remarque : durant la représentation du dendrogramme, on peut indiquer le nom des individus avec la commande :

```
a = dendrogram(Z,labels=list(data.index),color_threshold=seuil)
```

Q. 3 Analyser les classes. Combien d'individus contiennent-elles ?

Q. 4 Recommencer en utilisant une autre mesure de dissimilarité entre classe. Il peut être intéressant d'étudier graphiquement le résultat de la partition. Pour cela, on peut par exemple représenter les villes selon leurs coordonnées géographiques et avec une couleur associée à la classe. Pour cela, on peut utiliser els commandes suivantes :

```
Coord = data_temperature.loc[:,['Latitude','Longitude']].values
# Cette ligne permet d extraire les coordonnees
plt.scatter(Coord[:, 1], Coord[:, 0], c=methode.labels_, s=20, cmap
    ='viridis')
# On place les points
nom_ville = list(data.index)
for i, txt in enumerate(nom_ville):
        # On place le nom des villes
        plt.annotate(txt, (Coord[i, 1], Coord[i, 0]))
```

2.3 Méthode des K-means

Q. 1 Utiliser la commande K-means pour le clustering.

Q. 2 Tracer la courbe d'inertie en fonction de nombre de clusters K. Trouver le K optimal en utilisant la règle de coude.

Q. 3 Afficher les partitions en fonction des coordonnées géographiques.

2.4 Comparaison des classifications

Q. 1 Comparer les classifications obtenues.

3 Réalisation algorithmique

Vous devez réaliser le programme permettant d'étudier l'algorithme k-means qui suivra le prototype suivant :

```
[clas,g2]=coalescence(x,K,g)
```

Ce programme applique la méthode K-means afin de regrouper les différents individus en K classes. Ce programme ne fonctionne que pour un ensemble d'individu élément de $\mathbb{R}^2$ (2 mesures pour chaque individu). On va utiliser la distance suivante entre $x_1, x_2 \in \mathbb{R}^2$:

$$d(x_1, x_2) = (x_1 - x_2)^t (x_1 - x_2)$$

Les paramètres :

- La variable `x` est une matrice qui doit contenir les différents individus de l'ensemble d'apprentissage rangés par colonne. Pour respecter le fonctionnement des commandes des librairies Python, le nombre de ligne est `N` et le nombre de colonne est 2.
- La variable `K` indique le nombre de classe que l'on désire.
- La variable `g` contient les centres de gravité initiaux rangés par colonne. `g` est donc une matrice de `K` lignes et 2 colonnes.

Le résultat :

- La variable clas est un vecteur qui indique le résultat de l'algorithme de coalescence. `clas[i]` indique le numéro de la classe de l'individu `x[:,i]`.
- La variable `g2` contient les centres de gravité finaux rangés par colonne.

Afin de tester votre programme de coalescence, vous allez recréer un ensemble d'apprentissage correspondant à l'acquisition de 2 classes d'individus chacun associé à une distribution gaussienne bidimensionnelle comme dans le premier exercice :

La première classe est composée de 128 individus et est associée à la fonction de densité de la loi normale $\mathcal{N}_2((2,2)^T, 2I_2)$:

$$f(\mathbf{x}) = f(x_1, x_2) = \frac{1}{4\pi} \exp\left\{ -\frac{1}{2} \left(\mathbf{x} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} \right)^t \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \left(\mathbf{x} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} \right) \right\}$$

La seconde classe est composée de 128 individus et est associée à la fonction de densité de la loi normale $\mathcal{N}_2((-4,-4)^T, 6I_2)$:

$$f(\mathbf{x}) = f(x_1, x_2) = \frac{1}{4\pi} \exp\left\{ -\frac{1}{2} \left(\mathbf{x} - \begin{bmatrix} -4 \\ -4 \end{bmatrix} \right)^t \begin{bmatrix} 6 & 0 \\ 0 & 6 \end{bmatrix} \left(\mathbf{x} - \begin{bmatrix} -4 \\ -4 \end{bmatrix} \right) \right\}$$

Q. 1 Développer l'algorithme de coalescence.

Q. 2 Appliquer l'algorithme de coalescence. Vous devez donc créer 2 centres de gravité initiaux (paramètres `g` de la fonction). Pour cela, vous allez choisir au hasard 2 individus de l'ensemble d'apprentissage qui constitueront les centres initiaux.

Q. 3 Étudier graphiquement le résultat de la partition.

Q. 4 Tester plusieurs fois et reporter les résultats.