TELECOM Nancy *(ESIAL)*
Maths Numériques

TP3 : *H-means / K-means*

Le but de ce TP est de coder (dans le même fichier module) quelques fonctions permettant de faire de la classification avec l'algorithme *H-means* (appelé en général *K-means*). Vous partirez du fichier disponible sur Arche qui contient :

1. la fonction `fabrique_donnees` permettant d'obtenir des jeux de données adéquats pour la méthode (la fonction `visu_donnees` permet de voir les points obtenus) ;
2. la fonction `scores_silhouette_naif` permettant de calculer les scores sihouettes des points associés à une partition en K classes ;
3. la fonction `visu_classes` permettant d'afficher graphiquement la partition obtenue pour un jeu de points (limitée pour du 2d avec 6 classes maximum) ;
4. la fonction `tirer_entiers_tous_distinct` que nous avons codée la semaine dernière ;
5. les signatures des fonctions à implémenter ;
6. une version vectorielle des fonctions `barycentrage` et `affectation` et la fonction `partition_init`.

Ce TP aura aussi pour but de voir l'apport de la vectorisation en numpy. Cependant pour gagner un peu de temps, les fonctions vectorisées sont fournies (des explications orales seront données). Attention : ne pas utiliser ces codes (même ceux avec vectorisation) en utilisation réelle mais ceux d'une bonne bibliothèque (comme scikit-learn).

Utilisation de `fabrique_donnees`

Voici un script (disponible sur Arche) qui explique comment utiliser cette fonction. On génére 4 clusters dont les centres sont les 4 points de coordonnées $(-1,-1),(-1,1),(1,-1),(1,1)$. À partir du centre k, on génére n_k points selon une loi normale centrée en C_k et d'écart type σ_k. Dans l'exemple, on choisit des écarts types assez petits vis à vis de l'écartement des différents centres, ce qui nous donne 4 clusters assez bien séparés (cf figure).

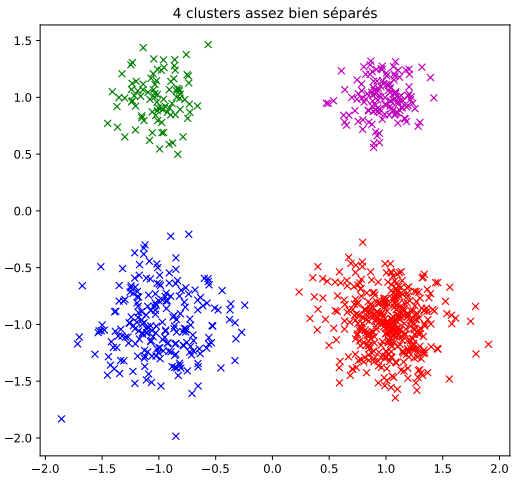
```
from hmeans import *
import numpy as np
import matplotlib.pyplot as plt

# on choisit de fabriquer 4 clusters de points
n_K = [220, 90, 400, 133] # nb de points pas cluster
mu_K = [[-1,-1],          # centre des 4 clusters
        [-1, 1],
        [1, -1],
        [1, 1]]
sigma_K = [0.3, 0.2, 0.25, 0.18] # écarts types pour la loi Normale

np.random.seed(5)  # on impose l'état du générateur aléatoire
X = fabrique_donnees(mu_K, sigma_K, n_K)

# affichage graphique
visu_donnees(X, n_K)
plt.title("4 clusters assez bien séparés")
plt.show()
```

Vous pouvez maintenant faire tourner ce script en modifiant les écarts types. Plus petits les clusters doivent être encore plus resserrés sur eux-même, plus grand une partie des points des différents clusters vont plus ou moins se mélanger.



Fonctions à implémenter

Rappel des tableaux utilisés. On part d'un tableau X de profil (n, p), i.e. n individus avec p caractères numériques. Les données utilisées seront fabriquées comme nous l'avons vu précédemment. Pour les centres/barycentres, on utilisera un tableau de profil (K, p). On utilise aussi un tableau *classe* de taille n où $classe[i]$ donne le numéro de classe du point i, etc.

1. Compléter la fonction `barycentrage(X, classe, K)` en utilisant l'algorithme donné en cours. Tester votre fonction à l'aide du script `test_barycentrage`.
2. Compléter la fonction `affectation(X, C)` en utilisant l'algorithme donné en cours. Tester votre fonction à l'aide du script `test_affectation.py`.
3. Comparaison avec les fonctions vectorielles. Utilisez le script `apport_vectorisation.py` pour (sur un exemple donné) voir les facteurs d'accélération obtenus. Ce script constitue aussi un deuxième test pour les deux fonctions précédentes. La vectorisation utilisée sera détaillée oralement. Sur ma machine et sur cet exemple précis, l'accélération est d'environ 150 pour l'affectation et de 30 pour les barycentres.
4. Compléter la fonction `hmeans(X, K, info=False)` qui implémente l'algorithme *H-means* en utilisant les fonctions `partition_init, barycentragev, affectationv`. Les itérations doivent aller jusqu'à la stabilisation complète de la partition, par exemple en utilisant le test :

   ```
   if np.all(classe == newclasse):
       break
   ```

   Lorsque `info` est "vrai", on affichera la suite des inerties (totales) obtenues.
5. Compléter la fonction `partition_qopt(X, K, nb_essais=10, info=False)` pour trouver une partition quasi-optimale (du moins espérée telle) des points X en K classes en utilisant l'algorithme *H-mean* `nb_essais` fois et en retenant la partition ayant la plus petite inertie totale. Lorsque `info` est "vrai", on affichera l'inertie (totale) obtenue pour chaque essai.
6. Écrire un script (peut-être à transformer en fonction ultérieurement) qui va, partant d'un jeu de données, tester la fonction précédente pour plusieurs valeurs de K. Afficher la courbe des inerties obtenues ainsi que celle des scores silhouettes pour chaque valeur de K. Faire des expériences avec différents jeux de données (plus ou moins bien séparés).