# K moyennes

INFORMATIQUE COMMUNE - TP n° 3.6 - Olivier Reynet

**À la fin de ce chapitre, je sais :**

- ☞ importer des données stockées dans un fichier de type csv
- ☞ coder l'algorithme k-means pour classifier des données non étiquetées
- ☞ visualiser un arbre de décision

## A K-means : classification non supervisée

On considère à nouveau un problème de **classification** mais cette fois-ci, le jeu de données ne dispose pas des étiquettes et on ne sait pas combien de classes existent a priori dans les données.

On suppose qu'on dispose de $n$ échantillons $\mathcal{E} = \{e_i, e_i \in \mathbb{R}^d, i \in [\![1, n]\!],\}$. **L'objectif est de créer une partition de $\mathcal{E}$ selon $k$ classes.**

**Algorithme 1** k moyennes (k-means)

```
1 : Fonction KMEANS(E, k, δ)
2 :     P ← {P_1, P_2, ..., P_k} une partition quelconque de E en k classes
3 :     tant que des échantillons changent de partition répéter
4 :         (b_1, b_2, ..., b_k) ← BARYCENTRE(P_1, P_2, ..., P_k)
5 :         pour chaque échantillon e de E répéter
6 :             Trouver la partition P_i la plus proche de e, ‖e − b_i‖² est minimale sur P
7 :             Ajouter e à la partition P_i trouvée
8 :     renvoyer (P_1, P_2, ..., P_k)
```

Cette section n'a pas pour but d'implémenter l'algorithme d'une manière très performante. On cherche avant tout à comprendre le fonctionnement de l'algorithme. Pour commencer, l'algorithme 1 nécessite la création d'une partition quelconque de $\mathcal{E}$ en $k$ classes. Dans ce but, on introduit l'algorithme de mélange de Knuth.

A1. Le mélange de Knuth (cf. algorithme 2) est un algorithme qui permet de générer une permutation aléatoire d'un tableau, c'est-à-dire qu'il génère chaque permutation avec la même probabilité. Écrire une fonction de signature `knuth_shuffle(tab: list)-> None` qui modifie le tableau passé en paramètre en le permutant selon l'algorithme 2.

**Solution :**

```
import random as rd
```

**Algorithme 2** Mélange de Knuth

```
1 : Fonction KNUTH_MIX(t)                          ▷ t est le tableau à permuter
2 :     pour i de n-1 à 1 répéter
3 :         j ← un entier aléatoire entre 0 et i
4 :         ÉCHANGER(a[j],a[i])
                                                   ▷ L'algorithme est en place
```

```
def knuth_shuffle(tab):
    n = len(tab)
    for i in range(n - 1, 0, -1):
        j = rd.randrange(0, i+1)
        tab[i], tab[j] = tab[j], tab[i]
```

A2. Quelle est la complexité de l'algorithme 2? On fera l'hypothèse que le tirage d'un entier aléatoire entre 0 et $i$ ne dépend pas de $i$ et s'effectue en un temps constant.

**Solution :** La complexité est linéaire en la taille du tableau à permuter, $\mathcal{O}(n)$. Il s'agit en effet d'une boucle répétée $n-1$ fois et les instructions du corps de boucle sont effectuées en un temps constant ne dépendant pas de $n$.

A3. Écrire une fonction `create_partitions(n:int , k: int)-> list[list]` qui génère une liste de $k$ parties aléatoires non vides de l'ensemble $[\![0, n-1]\!]$. On veillera à ne pas créer de partie vide : une partie contiendra au moins un élément. Par exemple, l'instruction `create_partitions(10, 5)` renvoie la liste `[[1, 5], [0], [7, 2], [3], [6, 9, 8, 4]]`. L'utilisation de la fonction `knuth_shuffle` est fortement conseillée.

**Solution :**

```
import random as rd

def create_partitions(n: int, k: int) -> list[list]:
    assert n >= k
    L = [i for i in range(n)]
    knuth_shuffle(L)
    P = []
    for i in range(k):
        P.append([L[i]])  # au moins un élément dans chaque partition
    for i in range(k, n):  # ajouter les autres éléments
        P[rd.randrange(0, k)].append(L[i])
    return P
```

A4. Écrire une fonction de signature `barycentre(partie,E)-> list[float]` qui renvoie le barycentre d'une partie `partie` de l'ensemble des échantillons `E`. `partie` est une liste d'indices correspondant à des échantillons de `E`. La fonction renvoie une liste de $d$ nombres flottants, si $d$ est la dimension de l'espace des échantillons.

**Solution :**

```
def barycentre(partie, E):
    barp = []
    n = len(E[0]) # dimension de l'espace des paramètres
    for k in range(n):  # pour chaque dimension
        bk = 0
        for p in partie:
            bk += E[p][k]
        barp.append(bk / len(partie))  # le barycentre de la dimension
    return barp  # le barycentre de la partie
```

A5. Écrire une fonction `barycentres(P: list[list[int]],data)-> list[list[float]]` qui calcule les $k$ barycentres de la partition `P` du jeu de données. Le résultat de cette fonction est une liste à $k$ éléments contenant des listes à $d$ éléments.

**Solution :**

```
def barycentres(P, data):
    b = []
    for p in P: # une partition
        b.append(barycentre(p, E)) # le barycentre de la partition
    return b
```

A6. Écrire une fonction `nearest_partition(e,B)-> int` qui renvoie l'**indice** de la partition dont le barycentre est le plus proche de l'échantillon `e`. `B` est la liste des barycentres obtenue à la question précédente. Il est **nécessaire** d'utiliser la distance euclidienne pour mesurer les distances entre les barycentres et l'échantillon (cf. TP K plus proches voisins).

**Solution :**

```
import math

def de(A: list[float], B: list[float]) -> float:
    assert len(A) == len(B)
    n = len(A)
    d = 0
    for i in range(n):
        d += (A[i] - B[i]) ** 2
    return math.sqrt(d)

def nearest_partition(p, B):
    index = 0
    d = de(p, B[0])
    for j in range(len(B)):
        dpB = de(p, B[j])
        if dpB < d:
            index, d = j, dpB
    return index
```

A7. Écrire une fonction de signature `kmeans(E, k)` qui implémente l'algorithme K-means (cf. algorithme 1).

**Solution :**

```
def kmeans(E, k):
    n = len(E)
    assert n >= k
    P = create_partitions(n, k)
    while True:
        B = barycentres(P, E)
        new_P = [[] for _ in range(k)]
        for i in range(n):
            index = nearest_partition(E[i], B)
            new_P[index].append(i)
        if P == new_P:
            return P
        else:
            P = new_P
```

A8. En utilisant les fonctions d'importation du TP K plus proches voisins, tester cet algorithme sur les deux jeux de données `diabetes.csv` et `iris.csv`.

## B Compresser une image avec K-means

L'algorithme des k-moyennes peut être utilisé pour compresser des images. Le principe est le suivant : plutôt que d'enregistrer toutes les nuances de couleurs dans le fichier, on attribue à chaque pixel une couleur proche de l'originale mais dans une gamme limitée de couleurs possibles. Cette couleur proche est obtenue en regroupant des pixels selon leur proximité de couleur grâce à l'algorithme des k-moyennes. Par exemple, on peut décider de ne coder que 64 couleurs de l'image, ces 64 couleurs étant déterminées par l'algorithme. L'image pourra être compressée encore davantage à cause des répétitions de couleur qu'on fera apparaître dans le fichier.

B1. À l'aide de la bibliothèque skimage et de la fonction `imread` du module `skimage.io` (cf. ici), et des instructions suivantes, charger l'image `'plage.jpg'` dans un tableau Numpy.

```
image = io.imread('plage.jpg')
print(type(image))
io.imshow(image)
io.show()
```

B2. À l'aide de la fonction Numpy `reshape` et de l'attribut `shape` des tableaux Numpy, transformer les données de l'image rectangulaire en un vecteur de pixels. Ce vecteur aura donc autant d'éléments qu'il y a de pixels sur l'image. C'est un vecteur de pixels, c'est à dire un vecteur de trois entiers compris entre 0 et 255.

**Solution :**

```
# DIMENSIONS
rows = image.shape[0]
```

```
cols = image.shape[1]
# FLATTEN IMAGE
image = image.reshape(rows*cols, 3)
```

B3. Appliquer l'algorithme des k-moyennes au vecteur de la question précédente en utilisant l'algorithme `Kmeans` de la bibliothèque `sklearn.cluster` et en suivant les instructions suivantes. On choisira de créer 8, 16 ou 32 catégories de pixels.

```
classifier = KMeans(n_clusters=32)
classifier.fit(image)

# CREATE NEW IMAGE
compressed_image = classifier.cluster_centers_[classifier.labels_]
compressed_image = np.clip(compressed_image.astype('uint8'), 0, 255)
compressed_image = compressed_image.reshape(rows, cols, 3)

# SAVE AND SHOW IMAGE
io.imsave('plage_32.png', compressed_image)
io.imshow(compressed_image)
io.show()
```

Les valeurs des barycentres des catégories sont accessibles par `classifier.cluster_centers_[classifier.labels_]` si `classifier` est l'objet Python généré par la fonction `KMeans`. Ces couleurs permettent de construire la nouvelle image, de l'afficher à l'écran et la sauvegarder au format png.

B4. À partir de combien de couleurs votre œil perçoit-il la différence ? Observer la différence de taille des images. Quel est l'ordre du taux de compression ($t = \frac{s-c}{s}$, où $s$ est la taille originale et $c$ la taille compressée) ?

**Solution :** 32 couleurs suffisent pour satisfaire notre œil à première vue ! Les pixels pourraient être codés sur 5 bits. Le taux de compression obtenu est de l'ordre de 80%.

## C Des manchots et des arbres --→ HORS PROGRAMME

Les arbres de décision sont des algorithmes simples et puissant en apprentissage automatique. CART (cf. algoritghme 3) est un des meilleurs algorithmes d'arbre de décision. Il peut être utiliser pour classer des éléments et il est capable de gérer des paramètres d'entrées numériques continus ou discrets simultanément.

En se donnant quelques primitives de construction d'un arbre binaire (FEUILLE et NŒUD), une fonction permettant de tester l'homogénéité de la classe des données (HOMOGÈNES ?) et des fonctions de division du jeu de données (PARTAGER_AU_MIEUX et DIVISER_DONNÉES), il est possible de décrire simplement cet algorithme.

On se propose d'utiliser CART programmé dans la bibliothèque Scikit sur un jeu de données qui concerne les manchots. On peut charger ce jeu de données ainsi :

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

**Algorithme 3** Classification And Regression Trees (CART)

```
1 : Fonction CART(données)
2 :     si HOMOGÈNES ?(données) alors renvoyer FEUILLE(Étiquette(données))
3 :     meilleur_partage ← PARTAGER_AU_MIEUX(données)
4 :     si meilleur_partage est vide alors renvoyer FEUILLE(Étiquette_majoritaire(données))
5 :     gauche, droite ← DIVISER_DONNÉES(données,meilleur_partage)
6 :     fils_gauche ← CART(gauche)
7 :     fils_droite ← CART(droite)
        renvoyer NŒUD(meilleur_partage, fils_gauche, fils_droite)
```

```
from sklearn.preprocessing import OrdinalEncoder
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_selector as selector

import seaborn as sns
df = sns.load_dataset('penguins')
```

C1. Faire afficher la description du jeu de données à l'aide des instructions suivantes. Contient-il uniquement des paramètres numériques continus ?

```
print(df)
print(df.describe())
df.info()
```

Pour que l'algorithme puisse travailler, il est nécessaire que toutes les données soient numériques. On choisit de convertir les données textuelles (des catégories) en entiers à l'aide d'un encodeur ordinal comme suit :

```
categorical_columns_selector = selector(dtype_include=object)
categorical_columns = categorical_columns_selector(df)
categorical_columns.pop(0) # remove species
print(categorical_columns)
data_categorical = df[categorical_columns]
encoder = OrdinalEncoder()
data_encoded = encoder.fit_transform(data_categorical)
print(data_encoded)
df[categorical_columns] = data_encoded
print(df.describe())
df=df.dropna() # remove NaN values
```

C2. Faire afficher sur un graphique la dispersion des paramètres de ce jeu de données à l'aide des instructions suivantes :

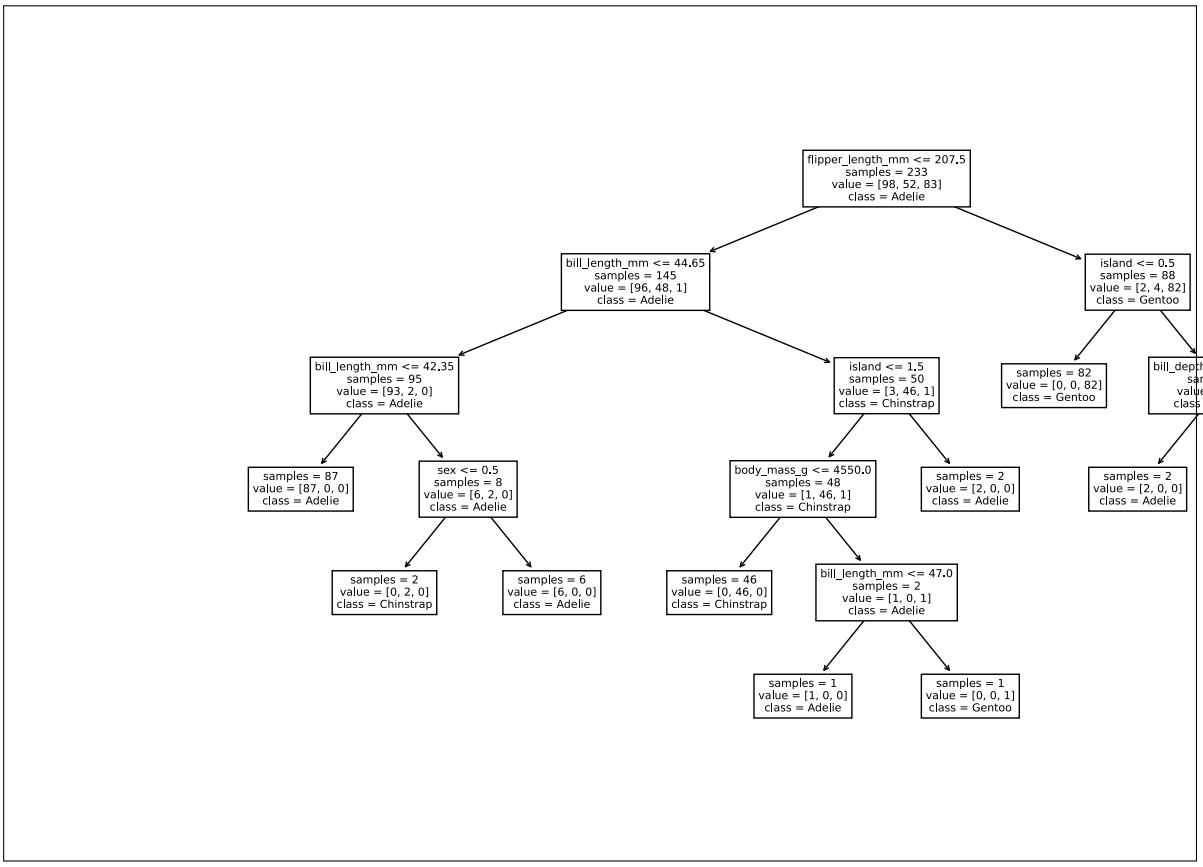
```
# Matrice de dispersion des paramètres / SCATTER MATRIX
sns.pairplot(data=df, hue='species', palette='mako')
plt.show()
```

C3. À l'aide de la fonction `DecisionTreeClassifier` de Scikit, entraîner un arbre de décision sur le jeu de donner. On limitera la profondeur de l'arbre à 5 niveaux. À l'aide du code suivant, afficher la matrice de confusion du classificateur et l'arbre de décision obtenu.

```
# CREATE DATASETS
X = df[['island', 'bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', '
    body_mass_g', 'sex']].values
y = df['species'].values
print('X -> ', X)
print('Y -> ', y)

# SPLIT DATASETS -> TRAIN/TEST
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=0)
print('Train set shape -> ', X_train.shape, y_train.shape)
print('Test set shape -> ', X_test.shape, y_test.shape)

# MACHINE LEARNING
classifier = DecisionTreeClassifier(max_depth=5)
classifier.fit(X_train, y_train)
print(classifier)

# TEST CLASSIFIER / COMPARE TRUE AND PREDICTION
predicted = classifier.predict(X_test)
print('Prediction Accuracy Score (%) :', round(accuracy_score(y_test, predicted)
     * 100, 2))

# CONFUSION MATRIX
cm = confusion_matrix(y_test, predicted)
print(cm)
labels = np.unique(df['species'].values)
ax = sns.heatmap(cm / np.sum(cm), annot=True, cmap='mako', xticklabels=labels,
    yticklabels=labels)
plt.show()

# PLOT TREE
_, ax = plt.subplots(figsize=(15, 8))
_ = plot_tree(classifier,
              feature_names=['island', 'bill_length_mm', 'bill_depth_mm', '
                  flipper_length_mm', 'body_mass_g', 'sex'],
              class_names=classifier.classes_,
              impurity=False, ax=ax)
plt.savefig('penguins_tree.svg')
plt.show()
```

**Solution :**

```
# Results
Prediction Accuracy Score (%) : 98.0
[[48  0  0]
[ 1 15  0]
[ 1  0 35]]
```


flipper_length_mm <= 207.5
samples = 233
value = [98, 52, 83]
class = Adelie
bill_length_mm <= 44.65
samples = 145
value = [96, 48, 1]
class = Adelie
island <= 0.5
samples = 88
value = [2, 4, 82]
class = Gentoo
bill_length_mm <= 42.35
samples = 95
value = [93, 2, 0]
class = Adelie
island <= 1.5
samples = 50
value = [3, 46, 1]
class = Chinstrap
samples = 82
value = [0, 0, 82]
class = Gentoo
bill_depth_mm <= 18.95
samples = 6
value = [2, 4, 0]
class = Chinstrap
samples = 87
value = [87, 0, 0]
class = Adelie
sex <= 0.5
samples = 8
value = [6, 2, 0]
class = Adelie
body_mass_g <= 4550.0
samples = 48
value = [1, 46, 1]
class = Chinstrap
samples = 2
value = [2, 0, 0]
class = Adelie
samples = 2
value = [2, 0, 0]
class = Adelie
samples = 4
value = [0, 4, 0]
class = Chinstrap
samples = 2
value = [0, 2, 0]
class = Chinstrap
samples = 6
value = [6, 0, 0]
class = Adelie
samples = 46
value = [0, 46, 0]
class = Chinstrap
bill_length_mm <= 47.0
samples = 2
value = [1, 0, 1]
class = Adelie
samples = 1
value = [1, 0, 0]
class = Adelie
samples = 1
value = [0, 0, 1]
class = Gentoo