Apprentissage non supervisé - Classification
Algorithme des k-moyennes de Lloyd

1 IA - Apprentissage non supervisé - Classification

📚 Dans un **apprentissage supervisé** (algorithme k-NN par exemple), l'algorithme formule une prédiction quant à l'appartenance d'une donnée à des classes prédéterminées (les données étiquetées).

📚 Au contraire, dans un **apprentissage non supervisé**, l'algorithme cherche à diviser des données non étiquetées en k partitions non déterminées a priori.
Il s'agit d'un problème de ***classification*** (de regroupement, de partitionnement, ***clustering*** en anglais).

📚 En intelligence artificielle, plus précisément en ***apprentissage automatique***, la méthode des k-moyennes ou algorithme de Lloyd est une méthode d'***apprentissage non supervisé***.

1.1 Principe de l'algorithme

1.1.1 Structure des données

On dispose d'un ensemble E d'objets o de même structure (un objet est représenté par un tableau de nombres). On considère qu'il est possible de définir une distance (ou indice de similarité) entre ces objets.

1.1.2 Moment d'inertie d'une famille de p objets par rapport à un centre c

Étant donnés une famille $(o_0, \cdots o_{p-1})$ d'objets et c un objet de même structure, le moment d'inertie de la famille par rapport à c est défini par $\sum_{i=0}^{p-1} \|o_i - c\|^2$.
Cette quantité positive fournit une ***mesure de la dispersion*** de la famille par rapport à c.

1.1.3 💡 Principe

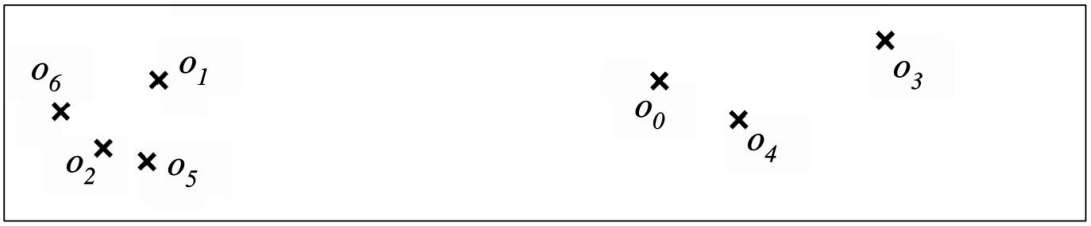
A partir d'un ensemble de données (ou objets) et d'un entier k, le problème est de diviser les données en k groupes, souvent appelés ***clusters***, de façon à minimiser l'inertie totale des k groupes de données par rapport à leurs k centres.

1.1.4 Inertie totale d'une partition par rapport à k centres

Soient $o = (o_0, \cdots o_{n-1})$ une famille de n objets à classifier et $k > 0$ le nombre de classes souhaité.
Classer ces objets consiste à déterminer pour chaque objet o_i une étiquette e_i ($0 \leq e\_i \leq k-1$) correspondant à l'une des classes.
Ainsi, la famille des n étiquettes $e = (e\_0, \cdots e\_n - 1)$ constitue un ***étiquetage*** des données.
Cet étiquetage induit une partition des objets en k classes : la classe C_j est l'ensemble des index i des objets o_i d'étiquette $e\_i = j$: $C\_j = \{i \in [0, n-1] \text{ tel que } e\_i = j\}$.
On considère également une famille $c = (c\_0, \cdots o\_k - 1)$ de k objets, n'appartenant pas nécessairement à la famille o, constituant les ***centres*** ou ***centroïdes*** des classes formées, et dont la détermination est explicitée dans l'algorithme des k-moyennes.

1.1.5 Exemple

On considère une famille $o = (o_0, o_1, o_2, o_3, o_4, o_5, o_6)$ de $n = 7$ objets (ci-dessous) à répartir en 2 classes.
Placer approximativement les centres (barycentres) c_0 et c_1 des deux groupes de points sur la figure.
En déduire les deux classes C_0 et C_1 constituées respectivement des index des points proches des barycentres c_0 et c_1.
En déduire l'étiquetage e (liste de $n = 7$ valeurs constituée de valeurs $i = 0$ ou $i = 1$ suivant que l'objet o_i appartient à la classe C_0 ou à la classe C_1.



Classe $C_0 = \{$ $\}$ Classe $C_1 = \{$ $\}$
Étiquetage : $e = ($ $)$

L'inertie totale de la partition est définie comme la somme, pour l'ensemble des classes, des moments d'inertie de chaque classe par rapport à son centre :
$\sum_{j=0}^{k}$ Moment d'inertie de la classe j par rapport à son centre c_j.

Plus précisément : $I(e, c) = \sum_{j=0}^{k} \sum_{i \in C_j} \|o_i - c_j\|^2$.

Le problème de classification consiste alors à minimiser l'inertie totale.
Il existe une ***heuristique*** classique pour ce problème, appelée **méthode des k-moyennes**. Il s'agit d'une heuristique car, selon l'initialisation, les résultats obtenus peuvent différer et rien ne garantit que le résultat final soit optimal.

1.1.6 Applications

L'exploration de données, ***data mining*** en anglais, chercher à constituer des groupes au sein de données afin d'appliquer des traitements ou des stratégies différentes en fonction des groupes.

2 Algorithme des k-moyennes de Lloyd

Objectif ✔️ : ***classer*** ou regrouper ou partitionner l'ensemble des données en k ***groupes*** ou ***clusters*** distincts, chaque groupe étant représenté par une donnée centrale dont les autres données sont proches.
Savoir faire ✏️ : écrire une bibliothèque de fonctions et utiliser la bibliothèque sklearn (https://scikit-learn.org/stable/index.html).

2.1 Notations - Définitions

— **Objet** o : tableau de dimension n caractérisant l'objet.
— **E** : ensemble d'objets ayant la structure de l'objet o.

2.2 Principe de l'algorithme

Entrées :
- un ensemble E d'objets o_i ;
- Un nombre entier $k < 0$ de classes (ou groupes ou clusters).
- une fonction distance d définie pour les objets de E (cette fonction peut posséder différentes définitions, ce sera donc un paramètre des fonctions de la bibliothèque).

Sortie : Étiquetage : $e = (e_0, \cdots e_{n-1})$ avec $0 \leq ei = j \leq k-1$ où j est la classe du cluster affecté à chaque objet de E.

📚 **Algorithme** :

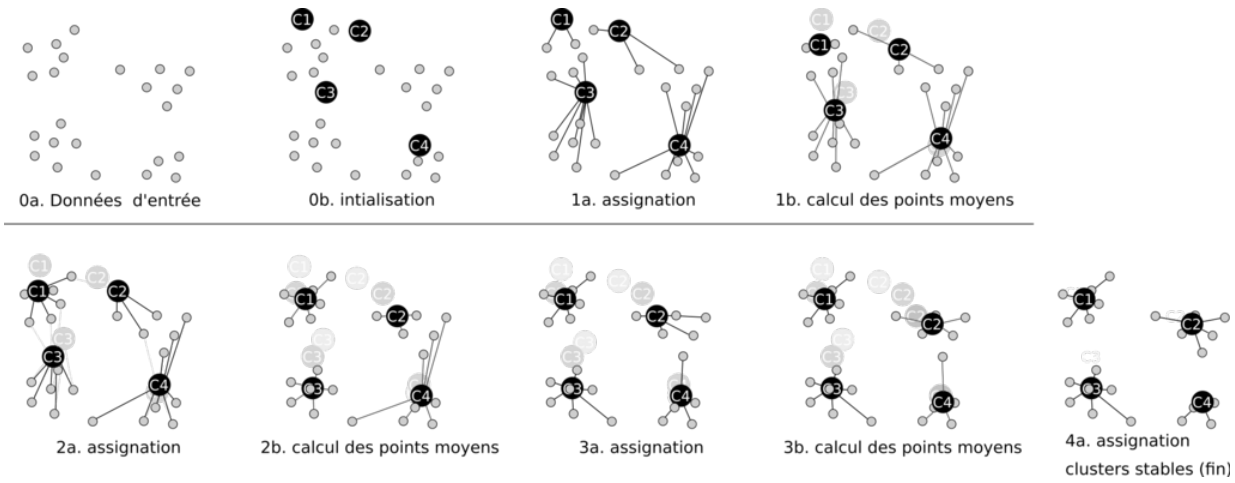
1. Initialisation aléatoire des centres des k classes et initialisation de l'étiquetage.
2. Itérations tant que l'étiquetage est modifié :
   - étiquette i affectée aux objets dans le groupe du centre c_i le plus proche ;
   - calcul du barycentre c_i des objets de chacun des k groupes.

Remarque : l'implémentation des fonctions ci-dessous ainsi que leurs paramètres sont choisis de façon à être compatibles avec la bibliothèque scikit learn.

💡 Lors de l'initialisation, ces centres sont choisis au hasard parmi toutes les données.
Chaque itération consiste à affiner le partitionnement : le ***barycentre*** de chacun des k groupes est recalculé (d'où le nom k-moyennes).
Les données sont alors regroupées en utilisant les nouveaux centres.
Rq : un barycentre ne coïncide pas nécessairement avec une donnée.

L'initialisation des centres conditionne le résultat final : suivant le choix initial des centres, l'algorithme construit des groupes différents et le ***résultat n'est pas nécessairement optimal***.
On parle d'***optimum local*** lorsque ***l'algorithme converge vers des minima locaux*** en fonction de l'initialisation.
En pratique, on exécute l'algorithme plusieurs fois avec des initialisations aléatoires des centres.

2.3 Illustration de l'algorithme



3 I - Écriture d'une bibliothèque de fonctions

3.1 Bibliothèques

```
from _validationkmoyennes import *
from random import random, randrange, randint, seed
```

3.2 Étape 1 - Calcul des distances entre deux objets

Etant donnés deux objets o_1 et o_2 représentés par des vecteurs dans un espace à n dimensions de coordonnées respectivement (x_i) et (y_i), la distance utilisée dans la suite est le carré de la distance euclidienne : $d(o_1, o_2) = \sum_{i=1}^{n} (x_i - y_i)^2$

✏️ Ecrire une fonction `d_Euclide2(o1, o2)` renvoyant le carré de la distance euclidienne entre les objets o1 et o2.
Rappel : ces objets sont des vecteurs de dimension inconnue a priori (mais identique pour les deux objets).
La fonction sum de python n'est pas autorisée.

```
def d_Euclide2(o1, o2):
    """
    Paramètres :
        o1, o2 : 2 objets = 2 listes ou tuples de coordonnées
    Renvoie :
        Distance euclidienne entre o1 et o2
    """


# Tests distances sur plusieurs couples (o1, o2)
for o1, o2 in [((0,0),(0,1)), ((0,0),(1,0)), ((0,0),(1,1)), ((0,0),(-2,0))]:
    print(f'Distance entre {o1} et {o2} : {d_Euclide2(o1,o2)}')
```

3.3 Étape 2 - Barycentre de plusieurs objets

✏️ Écrire une fonction `barycentre(L)` renvoyant la liste des coordonnées de l'isobarycentre d'une liste, non vide, d'objets.

L'isobarycentre de n objets o_i à p coordonnées $o_i \left| \begin{matrix} x_{i0} \\ \vdots \\ x_{ip} \end{matrix} \right.$ a pour coordonnées $\left| \begin{matrix} \overline{x}_0 = \dfrac{\sum_{i=0}^{n} x_{i0}}{n} \\ \vdots \\ \overline{x}_p = \dfrac{\sum_{i=0}^{n} x_{p0}}{n} \end{matrix} \right.$.

```
def barycentre(L):
    """
    Paramètres :
        L : liste d'objets (objet = liste ou tuple de coordonnées)
    Renvoie:
        Isobarycentre des objets = liste des moyennes pour chaque coordonnée
    """


# Tests barycentre avec 2 exemples (calculables de tête)
tests = [[(-1, 0), (0, 1), (1, 0), (0, -1)], [(1, 1),(3, 1)]]
for L in tests:
    print(barycentre(L))
```

📝 *module random*

`random.randint(a, b)`
Renvoie un entier aléatoire entier N tel que $a \leq N \leq b$.

`random.randrange(start, stop[, step])`
Renvoie un entier N choisi aléatoirment dans l'intervalle range(start, stop, step) (donc $start \leq N < stop$).

3.4 Étape 3 - Tirage aléatoire de k centres

✏️ Ecrire une fonction `init_centres(L, k)` renvoyant la liste de k objets tirés aléatoirement dans la liste L de n objets.

```
# Test init_centres : tirage au hasard de k centres dans L
L = [(-1, 0), (0, 1), (1, 0), (0, -1)]
k = 2
init_centres(L, k)
```

✏️ Ecrire une fonction `init_centres(L, k)` renvoyant la liste de k objets **distincts** tirés aléatoirement dans la liste L de n objets.
Pour s'assurer que tous les tirages sont distincts, après chaque tirage d'un objet o, on renouvelle le tirage tant que l'objet o figure dans la liste des objets tirés.

```
def init_centres(L, k):
    """
    Paramètres :
        L : liste d'objets (objet = liste ou tuple de coordonnées)
        k : nombre de groupes/clusters/classes
    Renvoie:
        Lcentres : liste de points choisis aléatoirement dans L comme centres
    """
```

```
# Test init_centres
L = [(-1, 0), (0, 1), (1, 0), (0, -1)]
init_centres(L, 2)
```

👁 *Variable Lcentres*

Dans toute la suite, *Lcentres* désigne la liste de k points représentant les k centres.

3.5 Étape 4 - Index du centre le plus proche

✏️ Écrire une fonction `index_centre_plus_proche(o, Lcentres, fct_distance)` renvoyant l'index de l'élément de *Lcentres* le plus proche de l'objet o.

```
def index_centre_plus_proche(o, Lcentres, fct_distance):
    """
    Paramètres :
        o :           objet de E (liste ou tuple de coordonnées)
        Lcentres :    liste des centres des clusters
        fct_distance: fonction utilisée pour calculer la distance
    Renvoie:
        imin : index de l'élément de Lcentres le plus proche de l'bjet o
    """
```

```
# Test index_centre_plus_proche
L = [(-2, 0), (0, 1), (5, 0), (0, -1)]
o = (3, 0)
for e in L:
    print(f"Distance d({o}, {e})\t= {d_Euclide2(o,e)}")
imin = index_centre_plus_proche(o, L, d_Euclide2)
print(f"Objet le plus proche : {L[imin]} => index = {imin} dans L = {L}")
```

👁 *Variables E et Letiquettes*

Dans toute la suite :
- E désigne la liste des coordonnées de tous les objets à classer ;
- *Letiquettes* désigne la liste, de même longueur que E, telle que Letiquettes[i] = index j du centre le plus proche de l'objet $o_i = E[i]$ dans la liste *Lcentres*.

3.6 Étape 5 - Étiquetage à centres fixés

✏️ Écrire une fonction `etiquetage(E, Letiquettes, Lcentres, fct_distance)` qui parcourt les objets de E et vérifie que l'étiquette d'un objet est bien l'index du centre le plus proche et qui modifie l'étiquette le cas échéant pour que ce soit le cas ; elle renvoie la valeur True si une étiquette (au moins) a été modifiée, False sinon.

```
def etiquetage(E, Letiquettes, Lcentres, fct_distance):
    """
    Paramètres :
        E :           liste d'objets (objet = liste ou tuple de coordonnées)
        Letiquettes : etiquette[i] = index j du centre le plus proche de E[i]
        Lcentres :    liste des éléments choisis comme centres des clusters
        fct_distance: fonction utilisée pour calculer la distance
    Renvoie:
        maj : booléen (True = 1 étiquette au moins modifiée)
    """
```

```
# Test étiquetage
E = [(-2, 0), (-3, 1), (5, 0), (6, -1)]
Lcentres = [(-4, 1), (6,1)]
Letiquettes = [0, 0, 0, 0]
etiquetage(E, Letiquettes, Lcentres, d_Euclide2)
for o, e in zip(E, Letiquettes):
    print(f"Objet {o}\t: centre le plus proche {Lcentres[e]}\t=> étiquette {e}")
print('Etiquetage : ', Letiquettes)
```

3.7 Étape 6 - Partition

✏️ Écrire une fonction `partition(E, Letiquettes, k)` renvoyant un dictionnaire de la forme classe : liste des objets de la classe (chaque objet de E est affecté à un cluster dont le centre est l'un des k centres ; il y a autant de clusters que de centres).
Dans l'exemple ci-dessus avec 7 objets répartis en 2 classes, les objets o_1, o_2, o_5, o_6 sont proches du centre c_0 et les objets o_0, o_3, o_4 sont proches du centre c_1 : la fonction partition devrait renvoyer un dictionnaire de la forme : $\{0 : [o_1, o_2, o_5, o_6], 1 : [o_0, o_3, o_4]\}$.

```
def partition(E, Letiquettes, k):
    """
    Paramètres :
        E :           liste d'objets (objet = liste ou tuple de coordonnées)
        Letiquettes : etiquette[i] = index j du centre le plus proche de E[i]
        k :           nombre de groupes/clusters/classes
    Renvoie:
        classe : dictionnaire {classe : liste des objets de la classe}
    """
```

```
# Test partition
E = [(-2, 0), (-3, 1), (5, 0), (6, -1)]
Lcentres = [(-4, 1), (6,1)]
Letiquettes = [0, 0, 0, 0]
etiquetage(E, Letiquettes, Lcentres, d_Euclide2)
for o, e in zip(E, Letiquettes):
    print(f"Objet {o}\t: centre le plus proche {Lcentres[e]}\t=> étiquette {e}")
print('Partition : ', partition(E, Letiquettes, 2))
```

3.8 Étape 7 - Calcul des centres à étiquetage fixé

✏️ Écrire une fonction `centres(E, Letiquettes, k)` renvoyant la liste des centres calculés à partir d'un étiquetage *Letiquettes* donné : la fonction réalise la partition des objets de E en fonction de l'étiquetage puis parcourt les différentes classes afin de calculer le barycentre de chaque classe. Si une classe est vide, on lui affecte comme centre

un élément de E choisi aléatoirement.

```
def centres(E, Letiquettes, k):
    """
    Paramètres :
        E :           liste d'objets (objet = liste ou tuple de coordonnées)
        Letiquettes : etiquette[i] = index j du centre le plus proche de E[i]
        k :           nombre de groupes/clusters/classes
    Renvoie:
        Lcentres: liste de points choisis aléatoirement dans E comme centres
    """
    # classe : dictionnaire {classe : liste des objets dans la classe}

    # Détermination des nouveaux centres


# Test centres
E = [(-2, 0), (-3, 1), (5, 0), (6, -1)]
Lcentres = [(-4, 1), (6,1)]
Letiquettes = [0, 1, 1, 0]
centres(E, Letiquettes, 2)
```

3.9 Étape 8 - k-moyennes

✏️ Écrire la fonction `k_moyennes(E, Lcentres, max_iter, fct_distance)` qui a pour paramètres l'ensemble E des objets à classer, une liste initiale de k centres (correspondant aux k classes) et un entier *max_iter*, nombre maximum d'itérations autorisé.
La fonction initialise une liste d'étiquettes (choisies nulles ou aléatoirement entre 0 et k-1) puis procède à l'étiquetage tant que celui-ci n'est pas stable ou que le nombre maximum d'itérations n'est pas atteint, en recalculant les centres à chaque nouvel étiquetage.
La fonction renvoie la liste des étiquettes, la liste des centres et le nombre d'itérations.

```
def kmoyennes(E, Lcentres, max_iter, fct_distance):
    """
    Paramètres :
        E :           liste d'objets (objet = liste ou tuple de coordonnées)
        Lcentres :    liste des éléments initialements choisis comme centres des clusters (objets de E ou
        max_iter :    nombre maximum d'itérations
        fct_distance: fonction utilisée pour calculer la distance
    Renvoie:
        Letiquettes : etiquette[i] = index j du centre le plus proche de E[i]
        Lcentres :    liste des points calculés comme centres des clusters
        n_iter :      nombre d'itérations effectuées
    """
    # Détermination de k via le nombre de centres

    # Initialisation de toutes les étiquettes à 0 (ou au hasard entre 0 et k-1)

    # Initialisation du nombre d'itérations

    # Boucle jusqu'à obtenir un étiquetage stable ou bien interruption sur le nombre d'itérations


# Test k-moyennes
E = [(-2, 0), (-3, 1), (5, 0), (6, -1)]
Lcentres = [(-4, 1), (6,1)]
Letiquettes, Lcentres, n_iter = kmoyennes(E, Lcentres, 10, d_Euclide2)
print(f"Centres calculés : {Lcentres} en {n_iter} itération(s)")
for o, e in zip(E, Letiquettes):
    print(f"Objet {o}\t: centre le plus proche {Lcentres[e]}\t=> étiquette {e}")
```

4 II - Exemple simple - Visualisation graphique

On considère des fruits de forme arrondie dont les caractéristiques retenues sont :
- le poids ;
- le rapport grand axe / petit axe.

Les valeurs sont tirées au hasard dans des intervalles prédéterminés pour chaque fruit.
Dans cet exemple, le nombre initial de clusters est 3 : mandarines (points orangés), kiwi (points verts) et cerises (points rouges).

```
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
```

4.0.1 Initialisation des données

💻 👁️ La fonction `fruits()` disponible dans l'une des bibliothèques chargées renvoie :
- E, une liste de la forme [[poids fruit i, rapport fruit i],...] où poids et rapport sont tirés aléatoirement pour chaque type de fruit mais dans des intervalles différents pour les trois types (mandarines, kiwis, cerises) ;
- le dictionnaire des classes de la forme i : type de fruit ('mandarine', 'kiwi' ou 'cerise').

```
# Données initiales : fruits et dictionnaire des classes
E, classes = fruits()
print(f"Caractéristiques des fruits, poids, rapport : \n{E[0], E[50], E[100]}")
print(f"classes associées : \n{classes[0], classes[50], classes[100]}")
```

4.0.2 Représentation graphique

La fonction `graphe_fruits(E, classes, Letiquettes, k, nIter)` disponible dans l'une des bibliothèques chargées permet de tracer deux graphes permettant d'évaluer l'algorithme :
- le graphe des fruits (poids en abscisses, rapports en ordonnée) avec leurs étiquettes réelles sous forme de couleurs ;
- le graphe des fruits avec les classes déterminées par l'algorithme des k-moyennes.

4.0.3 Programme

```
# Initialisation générateur de nombres pseudo-aléatoires
#seed(10)

# Nombre de clusters envisagés
k = 3  # Attention, maximum k = 5 (limitation dans le graphe_fruits)

# Données initiales : fruits et dictionnaire des classes
E, classes = fruits()

# Initialisation des centres
centresInit = init_centres(E, k)

# Calcul
Letiquettes, Lcentres, nIter = kmoyennes(E, centresInit, 10, d_Euclide2)

# Graphe
graphe_fruits(E, classes, Letiquettes, k, nIter)
```

4.0.4 Expérimenter

💻 👁️ Recopier le code ci-dessus et initialiser `centresInit` (ligne 11) à l'aide d'une liste de points judicieusement choisis, da façon à ce que les centres initiaux soient situés dans les différents clusters : utiliser les coordonnées des centres déterminées en survolant le graphe ci-dessus avec la souris (les coordonnées du pointeur s'affichent sous la barre d'outils des graphes).

```
# Choix humain des centres initiaux
# => succès de l'algorithme
```

```
# Nombre de clusters envisagés
k = 3  # Maximum = 5 (limitation dans le graphe_fruits : dic_coulk)

# Données initiales : E ("ensemble" de points) et dictionnaire des classes
E, classes = fruits() # classes : dictionnaire {pt i : index cluster}

# Initialisation manuelle des centres
centresInit = [E[i] for i in [30, 70, 200]]

# Calcul
Letiquettes, Lcentres, nIter = kmoyennes(E, centresInit, 10, d_Euclide2)

# Graphe
graphe_fruits(E, classes, Letiquettes, k, nIter)
```

5 III - Bibliothèque scikit learn

👁 *sklearn*

Algorithme des k-moyennes :
sklearn.cluster.KMeans
Syntaxe :
`y_pred = KMeans(n_clusters= k, n_init='auto').fit_predict(X)`
X est un tableau numpy de la forme `X = np.array([[x0, y0], [x1, y1], Ẽ)` contenant les coordonnées associées aux données.
y_pred est le tableau numpy contenant les index des classes.

On suppose qu'on dispose d'un tableau numpy, noté X, contenant les coordonnées associées aux données.
X est de la forme X = `np.array([[x0, y0], [x1, y1], Ẽ)`.
Le code ci-dessous permet alors de déterminer les groupes (clusters) associés à chaque classe et de tracer le nuage de points correspondant.

```
from sklearn.cluster import KMeans
y_pred = KMeans(n_clusters=k).fit_predict(X)
fig = plt.figure() plt.scatter(X[:, 0], X[:, 1], c=y_pred) plt.show()
```
abscisses = 1ère colonne du tableau = `X[:, 0]` ordonnées = 1ème colonne du tableau = `X[:, 1]`

6 IV - Bibliothèque scikit learn - Application à l'exemple précédent

6.1 Bibliothèques

💻 Exécuter le code.

```
import numpy as np
from sklearn.cluster import KMeans
```

6.2 Exemple précédent traité par sklearn

💻 👁 Utiliser la fonction KMeans de la bibliothèque sklearn.cluster pour partitionner les données créées par la fonction fruits() ; ne pas oublier de convertir E en tableau X numpy via la commande `X = np.array(E)`.

```
# Transformation des données précédentes (liste de listes en tableau numpy)
X = np.array(E)

# Nombre de groupes
k = 3
```

```
# Algorithme des k moyennes
y_pred = KMeans(n_clusters=k, n_init='auto').fit_predict(X)

# Représentation graphique
fig = plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title(f"Algorithme des k-moyennes (k = {k}) - sklearn")
plt.xlabel('Poids') ; plt.ylabel('Rapport grand axe / petit axe')
plt.show()
```

On peut constater l'efficacité de l'algorithme...