Chapitre 5

Algorithmes d'apprentissage

L'expression intelligence artificielle a été créée par les américains John McCarthy et Marvin Lee Minsky en 1959.
Il s'agit de faire exécuter par des machines des tâches habituellement exécutées par des humaines, qui nécessitent des capacités de raisonnement, de mémoire et d'apprentissage.
En particulier, on souhaite rendre une machine capable d'accéder à certaines connaissances sans avoir été programmée explicitement pour cela, autrement dit capable d'apprendre par elle-même.
L'expression apprentissage automatique (ou machine learning en anglais), a été créée par Arthur Samuel en 1959.
D'après Tony Michell (1988), on peut dire qu'une machine apprend si après certaines expériences elle a augmenté sa capacité à réaliser une certaine tâche.

I. Intelligence artificielle

Définition 5.1. Intelligence artificielle

Sous le terme d'intelligence artificielle se cachent un ensemble de techniques permettant à des machines d'accomplir des tâches et de résoudre des problèmes normalement réservés aux humains, comme par exemple reconnaître et localiser des objets dans une image

Depuis quelques années, on associe presque toujours l'intelligence aux capacités d'apprentissage : c'est grâce à l'apprentissage qu'un système intelligent, capable d'exécuter une tâche, peut améliorer ses performances avec l'expérience.

Définition 5.2. Apprentissage automatique

L'apprentissage automatique est un domaine de l'intelligence artificielle. Il s'agit de permettre à une machine de progresser, d'apprendre par elle-même à améliorer son fonctionnement dans un cadre de résolution d'un problème, mais sans jamais la programmer explicitement pour résoudre un problème.

À partir de données, un modèle est construit. Par exemple, pour apprendre à reconnaître un élément dans une image, on fournit à la machine des images avec la présence ou non de l'élément, et un modèle est élaboré. Ce modèle permet à la machine, lorsqu'on lui fournit une image nouvelle de dire si l'élément est présent ou non.
On distingue deux types d'apprentissage : **l'apprentissage supervisé** et **l'apprentissage non supervisé**.

Définition 5.3. Apprentissage supervisé

L'apprentissage supervisé consiste pour un opérateur à montrer à la machines des milliers voire des millions d'exemples étiquetés avec leur catégorie, qui permettront à la machine de déterminer elle-même les paramètres pertinents pour classer chaque objet dans la catégorie qui lui correspond. Une fois cette phase d'apprentissage terminée, la machine doit être capable de généraliser à des objets pas encore vus.

Définition 5.4. Apprentissage non supervisé

L'apprentissage non supervisé est plus ambitieux, mais il est aussi plus proche de notre propre modèle d'apprentissage, basé sur l'observation. Il consiste à faire en sorte qu'à partir d'un ensemble de données (non étiquetées) la machine soit capable de créer ses propres catégories, et si possible que ces catégories soient pertinentes pour nous.

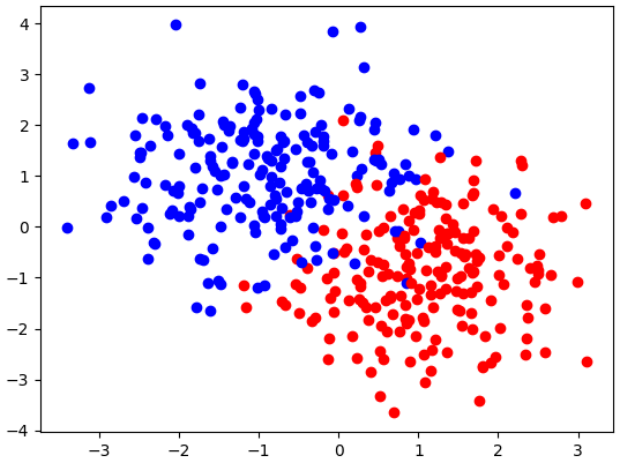
Dans la suite nous allons nous intéresser à deux algorithmes :

- l'algorithme des k plus proches voisins utilisé en apprentissage supervisé ;
- l'algorithme des k-moyennes utilisé en apprentissage non supervisé

II. Apprentissage supervisé : algorithme des k plus proches voisins (en anglais KNN pour k-nearest neighbors)

1. Principe KNN pour la classifications

À partir des caractéristiques fournies, on procède à un classement en décidant des caractéristiques d'un nouvel élément après l'examen des caractéristiques de ses « k plus proches voisins » . La notion de voisin devra être précisée et une difficulté est de bien choisir l'entier k.
Nous allons considérer un exemple simple, un ensemble de 200 points de coordonnées (x, y) dans le plan qui peuvent appartenir à deux catégories : bleu ou rouge. Nous fournissons à la machine le jeu de ces 200 données d'apprentissage.



Les données d'apprentissage sont constituées d'une liste de listes de format `[[p0, c0], [p1, c1], ...]` :

- `pi` représente les coordonnées du point i, données sous la forme d'une liste de deux éléments ;
- `ci` est une chaîne de caractères qui indique la couleur du point : `"b"` pour bleu, et `"r"` pour rouge.

Une fois ces données acquises, nous souhaitons que la machine attribue à un point P donné de

coordonnées (x, y) (et de catégorie inconnue) une catégorie rouge ou bleu.
Pour cela, nous utilisons la méthode des k plus proches voisins, qui consiste à déterminer la catégorie des k données d'entraînement les plus proches du point P(x, y), et à attribuer à ce point la catégorie majoritaire parmi les k voisins les plus proches.
L'algorithme des k plus proches voisins prend en entrée :

- la liste précédente des données d'apprentissage,
- un point p du plan, constitué d'une liste de ses deux coordonnées [x,y],
- un entier k le nombre de voisins considérés, qui sera choisi impaire (pour éviter les risques d'égalité puisqu'il y a deux catégories ici).

La notion de « k plus proches voisins » est prise en terme de distance entre le point p et les points des données d'apprentissage.

2. distance

Nous devons donc commencer par définir la distance. Nous utiliserons dasn ce cas la distance euclidienne (nous verrons d'autres exemples ou d'autres distances seront choisies).

Définition 5.5. distance euclidienne

La distance euclidienne entre les deux points du plan $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$ est définie :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

De manière générale, pour deux vecteurs de $\mathbb{R}^m$, $X = \begin{pmatrix} x_0 \\ \vdots \\ x_{m-1} \end{pmatrix}$, $Y = \begin{pmatrix} y_0 \\ \vdots \\ y_{m-1} \end{pmatrix}$, la distance euclidienne entre X et Y est :

$$d = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2 + \cdots + (x_{m-1} - y_{m-1})^2} = \sqrt{\sum_{k=0}^{m-1} (x_k - y_k)^2}$$

Exercice 5.1

1. Créer une fonction `distance2( p1 , p2 )` renvoyant la distance euclidienne entre les deux points placés en argument.
2. Créer une fonction `distancem( p1 , p2 )` qui généralise le problème dans le cas où les points sont en dimension m.

3. Algorithme des k plus proches voisins

Nous détaillons ici l'algorithme des k plus proches voisins pour déterminer la catégorie d'un point dont les coordonnées sont connues en utilisant un ensemble de données d'apprentissage :

1. Calculer les distances séparant l'ensemble des points P_i des données avec le point P dont on cherche à déterminer la catégorie.
   On obtiendra une liste de listes de deux valeurs `distance = [...., [di,ci], ...]`. Pour chaque liste, le premier élément est la distance d_i séparant le point P_i des données et le point P, et le deuxième élément est la couleur du point P_i.
2. Classer la liste des distances par ordre de distance croissante.
   On pourra utiliser la méthode `L.sort()` de python qui trie la liste `L` par ordre lexicographique du premier élément (c'est ce qu'on veut ici).
3. Parcourir les k plus proches voisins de P , et compter le nombre de voisins dans chacune des deux catégories.
4. Renvoyer la catégorie de P comme étant la majoritaire parmi ses k plus proches voisins.

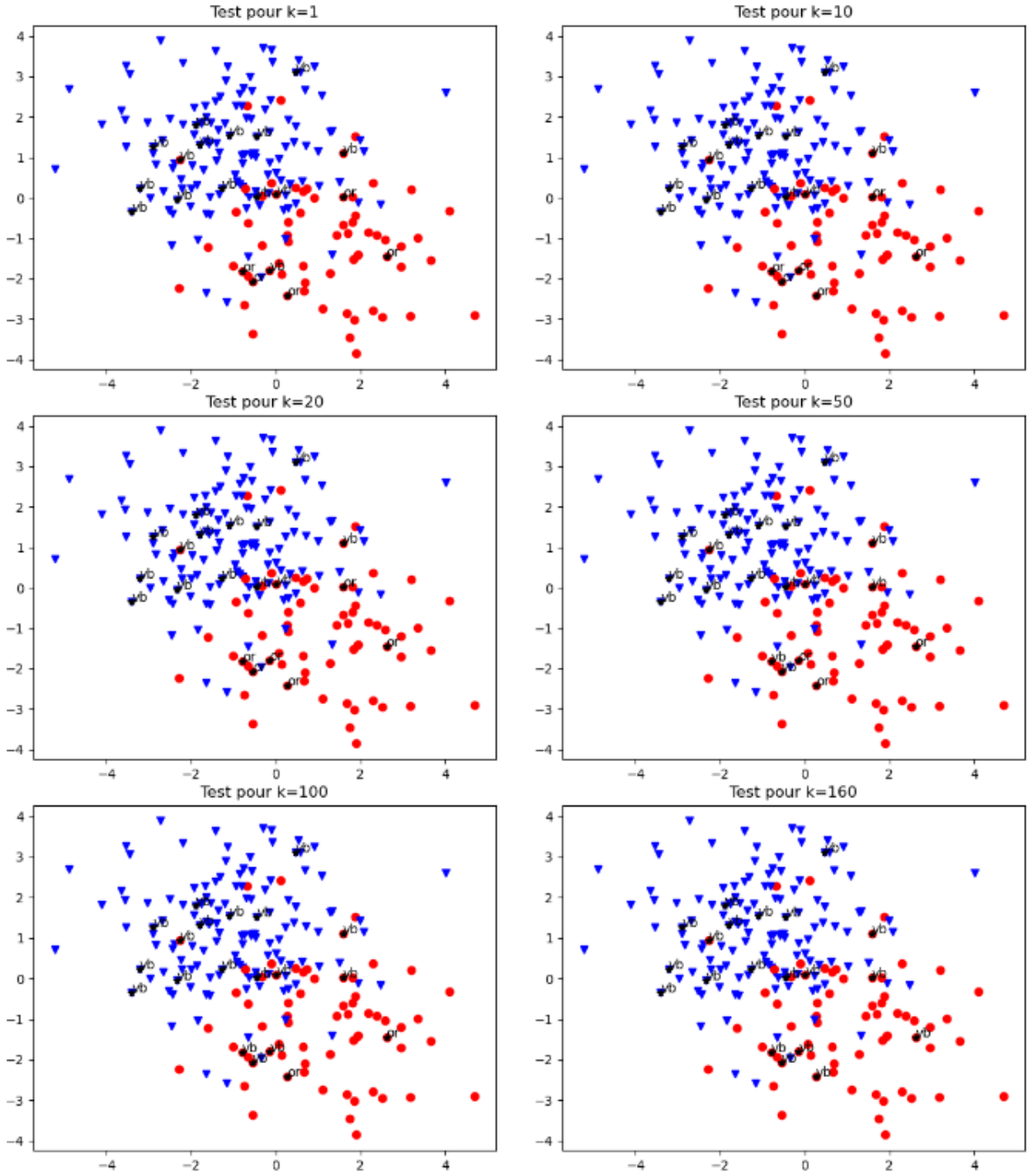
Exercice 5.2

A vous!

```python
def kPlusProchesVoisins(L : list, p : tuple , k : int) -> str :
    """
    Entrées : L : liste des données ( points Pi )
    p : point dont on cherche la catégorie
    k : nombre de voisins plus proches comptés
    Sortie : catégorie du point p
    """

    # Etape 1 : calcul de la distance de p à chaque point de L,
    # renseignée dans une liste D
    D               # initialisation de liste de liste de 2 éléments
    #[ distance du point Pi à p , couleur du point Pi ]
    for i in ......... :        # remplissage de la liste D

    # Etape 2 : tri selon la distance de p aux points de L


    # Etape 3 : détermination de la catégorie majoritaire des k plus proches voisins de p
```

4. Test de l'algorithme

Pour tester la qualité de l'algorithme et « voir » les résultats, dans le cas d'un apprentissage supervisé, nous pouvons séparer les données en deux groupes : des données d'apprentissage et des données qui permettent de tester la qualité de l'apprentissage.
Les 200 données ont été séparées en 180 données d'apprentissage et 20 données qui vont servir à tester l'algorithme.
L'algorithme a été testé en prenant différentes valeurs de k (c'est-à-dire en recherchant la catégorie majoritaire sur les k plus proches voisins).
Pour visualiser les résultats (sur les poly en noir et blanc), les données rouges sont des points, les données bleues sont des triangles.
Les points de test sont représentés par des étoiles noires, annotées de `or` si le point a été identifié comme rouge, et `vb` si le point a été identifié comme bleu.



5. Matrice de confusion

Pour aider à choisir un bon paramètre k, on va construire une matrice appelée **matrice de confusion** : À k fixé, on peut voir quelle couleur est attribuée à chacun des points test via l'algorithme des k plus proches voisins, et construire une matrice 2×2 de la forme $\begin{pmatrix} \text{RR} & \text{RB} \\ \text{BR} & \text{BB} \end{pmatrix}$ où

- RR (resp. BB) est le nombre de points de qui sont rouges (resp. bleus) et étiquetés comme tel par l'algorithme.
- RB (resp. BR) est le nombre de points de qui sont rouges (resp. bleus) et étiquetés en bleu (resp. rouge) par l'algorithme.

Pour calculer cette matrice, on doit disposer d'un ensemble de données à tester et de l'ensemble des

résultats attendus.
On compare alors les résultats attendus avec les résultats prédits sur les données à tester.
Pour cela, on utilise un ensemble de données classées. Ces données sont partagées en deux ensembles : l'un constituant les données d'apprentissages, et l'autre les données test.

```
def matrice_confusion(apprentissage, test, k ) :
    """
    Entrée :
    apprentissage : liste des données d ' apprentissage
    test : liste de liste des données utilisées pour le test
    k : nombre de voisins choisis pour identifier la catégorie
    Renvoi :
    mat : matrice de confusion 2 x2
    """
    mat = np . zeros ((2 ,2) ) # initialisation de la matrice de confusion
    for i in range ( ) : # parcours des données de test
        cpred =          # couleur prédite par application
                         # de l ' algorithme des k plus proches voisins
        cr =             # couleur réelle
        # remplissage de la matrice (4 cas possibles ) :
        if cr == " b " and cpred == " b " : # test [ i ] est étiqueté en bleu , et
                                           # la catégorie prédite est bleue
        elif

        elif

        else :

    return mat
```

Cette matrice de confusion peut aider à choisir la meilleure valeur de k. Pour un ensemble de $N = 500$ points (semblables aux données précédentes), 3/4 ayant servis de données d'apprentissage et 1/4 de données tests, on obtient les matrices de confusion M_k suivantes :

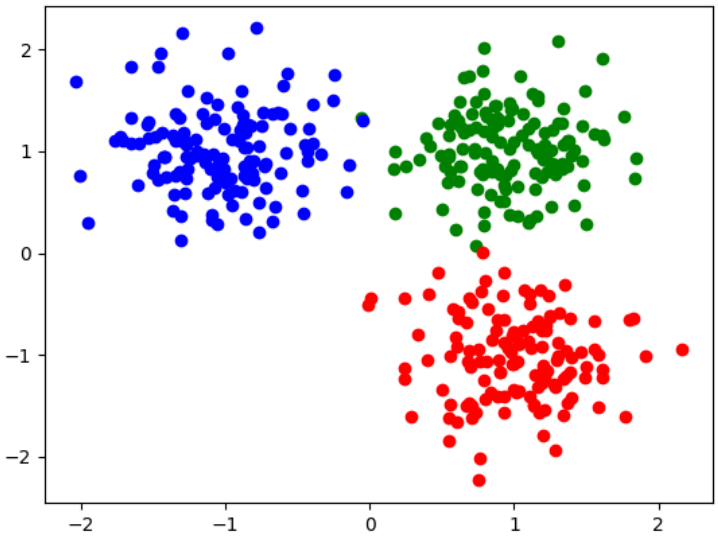
$$M_1 = \begin{pmatrix} 79 & 8 \\ 5 & 33 \end{pmatrix} \quad M_{50} = \begin{pmatrix} 81 & 3 \\ 3 & 38 \end{pmatrix} \quad M_{200} = \begin{pmatrix} 82 & 8 \\ 2 & 33 \end{pmatrix} \quad M_{300} = \begin{pmatrix} 84 & 41 \\ 0 & 41 \end{pmatrix}$$

III. Apprentissage non supervisé : algorithme des k-moyennes

Nous allons maintenant nous intéresser à un problème plus complexe : comment comprendre la structure des données sans aider (c'est à dire sans lui fournir des données d'apprentissage étiquetées) ? En pratique il s'agit pour la machine de regrouper les données proches au sein d'une même classe, à charge pour l'humain d'interpréter ensuite ce regroupement.

1. Définitions

Nous considérons un ensemble de points qui a l'allure ci-dessous. Nous observons que les points semblent se regrouper dans trois classes.



L'objectif est de permettre à la machine d'identifier ces trois classes. Nous allons tout de même l'aider en lui imposant le nombre de classes (3 ici).

Définition 5.6.

On note $C_0, ..., C_{k-1}$ les k classes, et on définit :

1. le barycentre μ_j de la classe C_j : $\mu_j = \dfrac{1}{\text{card}(C_j)} \displaystyle\sum_{x \in C_j} x$
2. le moment d'inertie m_j de la classe C_j : $m_j = \displaystyle\sum_{x \in C_j} \|x - \mu_j\|^2$

L'objectif de la machine est de minimiser la somme des moments d'inertie $\displaystyle\sum_{j=1}^{k} m_j$

2. Algorithme des k moyennes

Le calcul de la classification optimale, c'est-à-dire minimisant la somme des moments d'inertie, est un problème très coûteux en temps. Nous allons donc employer un algorithme glouton. Ce dernier nous assurera d'obtenir in fine une « bonne » solution, mais pas forcément la meilleure.

Algorithme des k moyennes

L'algorithme des k-moyennes consiste à réaliser la succession d'opérations suivantes :

1. on choisit aléatoirement k barycentres $\mu_0, ..., \mu_{k-1}$;
2. chacun des points du nuage est associé au barycentre $\mu_{j_{min}}$ le plus proche ;
3. on crée ainsi k classes $C_0, ..., C_{k-1}$;
4. on calcule les barycentres $\mu_0, ..., \mu_{k-1}$ de ces classes, qui remplacent les valeurs précédentes (des barycentres) ;
5. on reprend le calcul à partir de l'étape 2.

Nous admettons que cet algorithme converge, autrement dit qu'à partir d'une certaine étape les centres $\mu_0, ..., \mu_{k-1}$ ne se déplacent plus et donc que les classes $C_0, ..., C_{k-1}$ sont stabilisées.

ATTENTION : cet algorithme donne une configuration pour laquelle la somme des moments d'inertie est un minimum local, mais pas forcément le minimum global. Le résultat dépendant notamment du point de départ (choix aléatoire des centres).

3. Implémentation en Python

a. Calcul d'un barycentre

Nous allons commencer par écrire la fonction `barycentre` qui prend en entrée une liste de listes de 2 éléments représentant une liste de points représentés par leurs coordonnées, et qui renvoie les coordonnées du barycentre de ces points sous la forme d'une liste de deux valeurs.

```
def barycentre(L) :
    """
    Entrée : liste L de points , liste de listes de 2 éléments [[ x0 , y0 ] ,...]
    Renvoie : liste de 2 éléments : coordonnées du barycentre
    """
    xG, yG = 0, 0
    for i in range(len(L)):



    return [        ,       ]
```

b. Fonction k_moyenne

Puis on écrit la fonction `k_moyennes`, qui prend en entrées deux arguments :

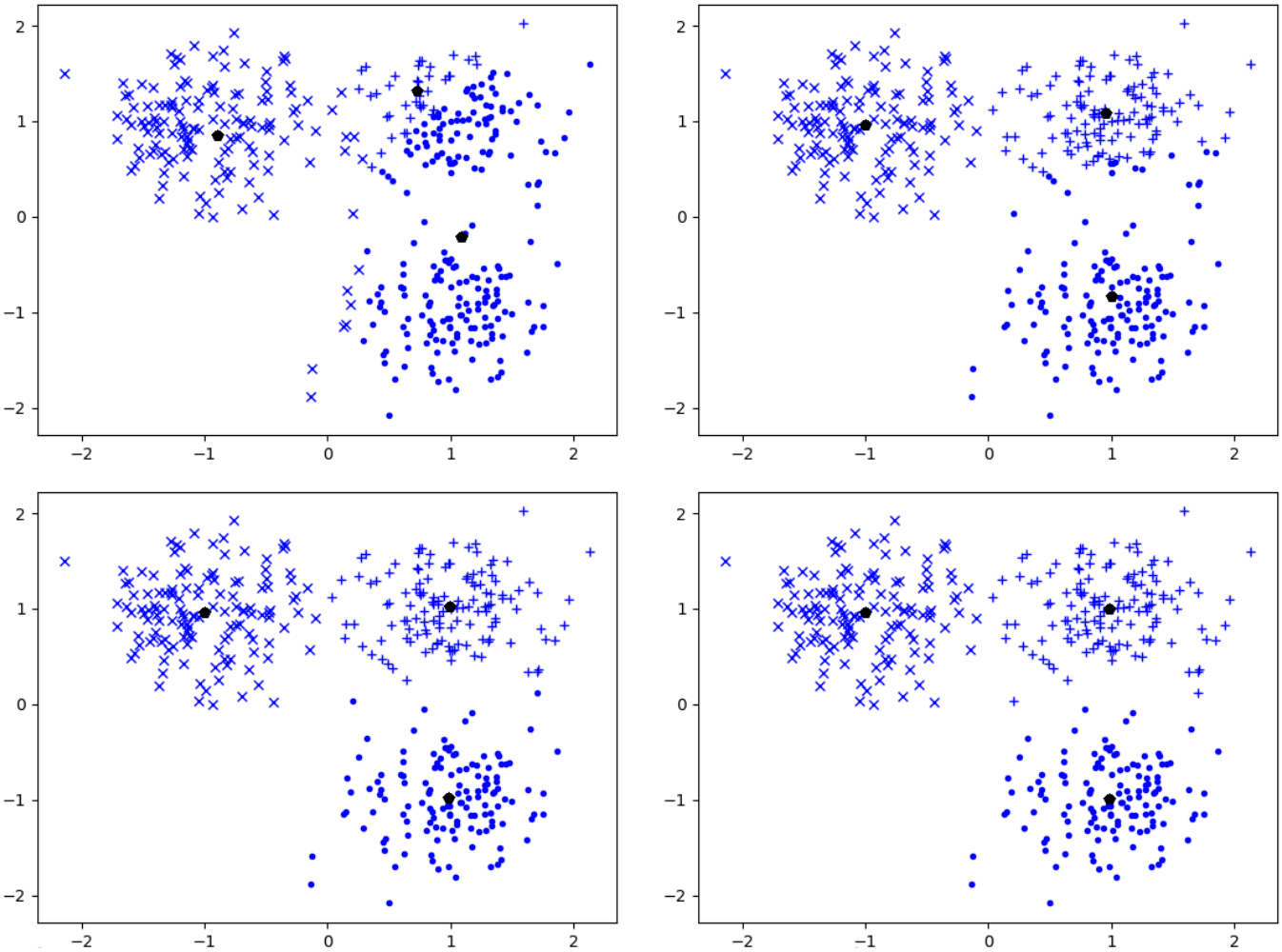
- la liste `donnees` qui contient l'ensemble des données non classées, qui sont des points représentés par des listes de deux éléments (les coordonnées des points),
- et un entier `k` qui est le nombre de classes que l'on impose à la machine.

Elle renvoie la liste `C` de `k` listes, où la liste `C[i]` est la liste des points de la classe C_i.

1. Pour la première étape, on pourra utiliser la fonction `randint(a,b)` qui renvoie un entier aléatoire entre `a` et `b` inclus.
2. Pour la deuxième étape, il faut parcourir l'ensemble des données. Pour chaque donnée, on calcule la distance entre la donnée et l'ensemble des barycentres choisis précédemment. On détermine la distance minimale et donc le barycentre le plus proche. La donnée est associée à cette classe.
3. Pour la troisième étape, il suffit de calculer les nouveaux barycentres.
4. Ces trois étapes sont répétées tant que les nouveaux barycentres sont différents des anciens.

```python
from copy import deepcopy # copie en profondeur d'une liste de listes
import numpy as np # pour avoir l'infini avec np.inf

def k_moyennes(donnees, k) :
    """
    Entrées : donnees : liste de listes de 2 éléments, des points à classer
    k : nombre de classes que l' on impose
    Sortie : C : liste qui est la partition de donnees en k parties non vides C0 ,... , Ck -1
    """
    # 1/ choix de départ des k centres : mu est la liste de k points des données
    # choisis aléatoirement
    mu = oldmu =[0 for i in range ( k ) ]
    while oldmu != mu :      # 4/ tant que ça n'a pas convergé
        C =[[] for i in range ( k ) ]    # liste de k listes(k classes) des points
                                         #   regroupés par classes
        # 2/ chacun des points i des données est associé au centre le plus proche
        for i in range ( ) : # parcours des données
            jmin =0      # rang du centre muj le plus proche du point i
            dmin = np.inf # initialisation de la distance min à l'infini
            for j in range(          ) : # parcours des centres à la recherche du
                                         # plus proche de la donnée i
                dij =           # distance entre le point i et le centre j
                if              : # on cherche le centre le plus proche de i


                                # on ajoute le point i à la classe de rang jmin
        # 3/ Calcul des barycentres de ces classes
        oldmu = deepcopy(mu) # garder en mémoire les anciens barycentres
                             # (pour pouvoir les comparer aux nouveaux calculés
                             # à partir des classes déterminées )
        mu =            # liste barycentres des classes déterminées à l'étape 2/
    return
```

Le nombre d'itérations nécessaires pour converger dépend des données, du nombre de classes imposées, et du premier choix aléatoire des trois barycentres. Sur le jeu de données ci-dessous, le nombre d'itérations nécessaire a varié entre 3 et 12 en l'ayant fait tourner une dizaine de fois.



On constate que l'algorithme converge vers des classes correctement identifiées, avec seulement quelques points mal identifiés. Les erreurs (nombres et natures) dépendent des données, du nombre de classes imposé, et du premier choix (aléatoire) des trois barycentres.