Algorithme de classification

Informatique tronc commun

Sylvain Pelletier

PSI - LMSC

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Introduction

- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (`KNN = k nearest neighbors`) et les k-moyennes (`k means`).
- Il s'agit de problème de classification : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- L'algorithme des KNN concerne le problème de L'APPRENTISSAGE SUPERVISÉ :
  - on dispose N objets sur lesquels on a effectué des mesures. (N grand).
  - Ces objets se répartissent selon T classes (T petit). On sait dans quel classe va chaque objet que l'on a mesuré.
  - On ajoute un nouvel objet sur lequel on a fait les mêmes mesures mais dont on ne connaît pas la classe.
  - De quelle classe est-il le plus proche ?

Choix de la distance

- Applications : commerce en ligne, médecine.

- Pour donner un sens à la classe la plus proche, cela suppose a `minima` de choisir une manière de mesurer les écarts entre les objets, ie d'avoir une distance sur les mesures associées à aux objets.
- Les mesures peuvent prendre diverses formes (nombres réels, vrai/faux, images, etc.). Certaines mesures ont plus d'importance que d'autres.
- Cela n'est pas du tout évident. Le choix de la distance est critique pour la classification.

Principe de l'algorithme

- Mathématiquement, on dispose donc d'un ensemble E assez complexe (l'ensemble des mesures), E est muni d'une distance d.
  On dispose d'un ensemble C de classe de cardinal T.
- On dispose de N éléments de $E \times C$ (les mesures faites sur les objets et les classes associées). On les notes $(x_j, c_j)_{j \in [\![1,N]\!]}$. Ainsi, x_j est la liste des mesures faites sur l'objet j, et c_j la classe associée.
- On considère de plus un autre élément y de E et on veut lui associer une classe.

Comme problème jouet, on considère donc que $E = \mathbb{R}^2$ et que l'on prend la distance euclidienne.

Principe de l'algorithme

- Mathématiquement, on dispose donc d'un ensemble E assez complexe (l'ensemble des mesures), E est muni d'une distance d.
  On dispose d'un ensemble C de classe de cardinal T.
- On dispose de N éléments de $E \times C$ (les mesures faites sur les objets et les classes associées). On les notes $(x_j, c_j)_{j \in [\![1,N]\!]}$. Ainsi, x_j est la liste des mesures faites sur l'objet j, et c_j la classe associée.
- On considère de plus un autre élément y de E et on veut lui associer une classe.

Comme problème jouet, on considère donc que $E = \mathbb{R}^2$ et que l'on prend la distance euclidienne.

Principe de l'algorithme

- Mathématiquement, on dispose donc d'un ensemble E assez complexe (l'ensemble des mesures), E est muni d'une distance d.
  On dispose d'un ensemble C de classe de cardinal T.
- On dispose de N éléments de $E \times C$ (les mesures faites sur les objets et les classes associées). On les notes $(x_j, c_j)_{j \in [\![1,N]\!]}$. Ainsi, x_j est la liste des mesures faites sur l'objet j, et c_j la classe associée.
- On considère de plus un autre élément y de E et on veut lui associer une classe.

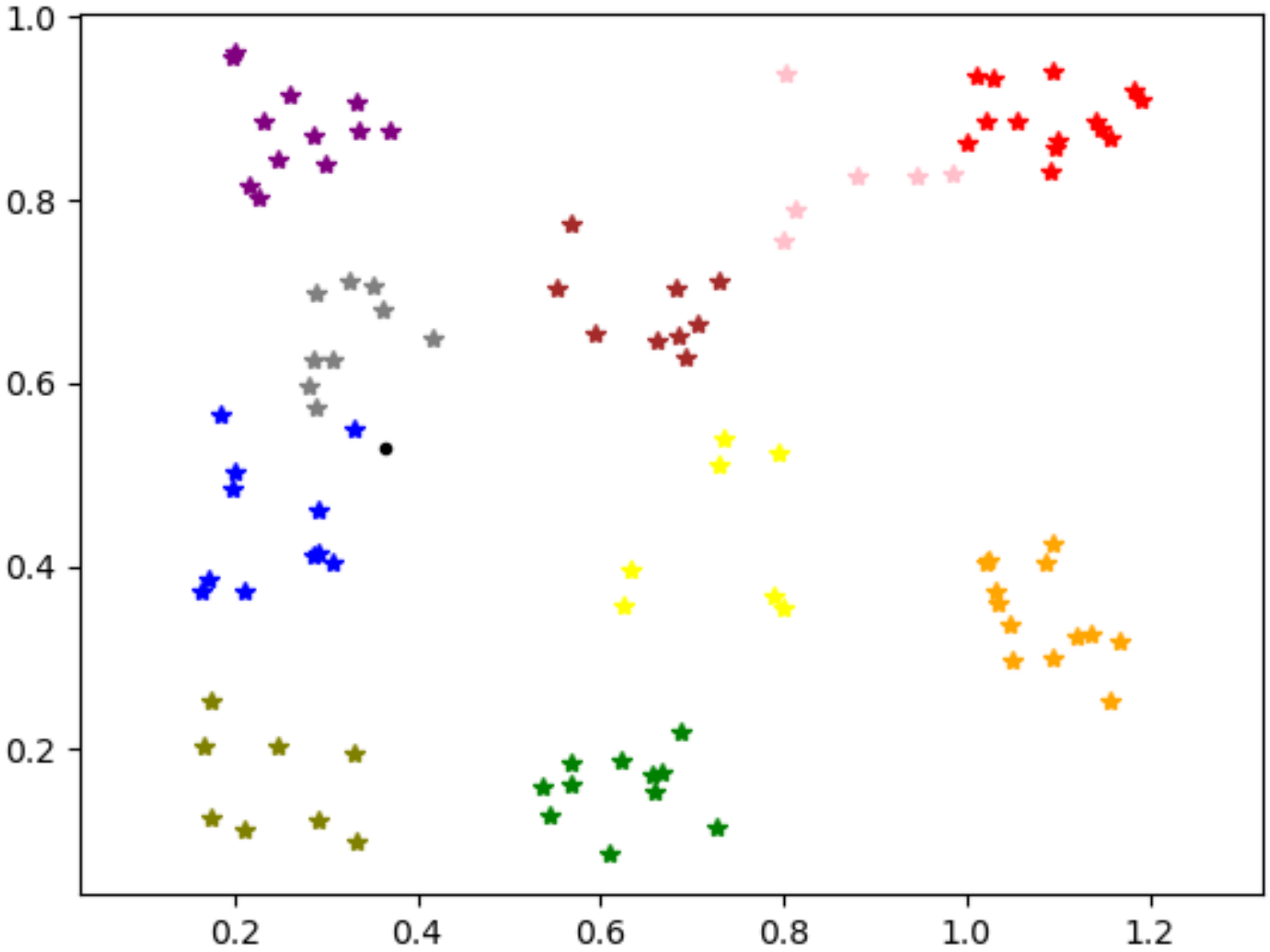
Comme problème jouet, on considère donc que $E = \mathbb{R}^2$ et que l'on prend la distance euclidienne.

Principe de l'algorithme

- Mathématiquement, on dispose donc d'un ensemble E assez complexe (l'ensemble des mesures), E est muni d'une distance d.
  On dispose d'un ensemble C de classe de cardinal T.
- On dispose de N éléments de $E \times C$ (les mesures faites sur les objets et les classes associées). On les notes $(x_j, c_j)_{j \in [\![1,N]\!]}$. Ainsi, x_j est la liste des mesures faites sur l'objet j, et c_j la classe associée.
- On considère de plus un autre élément y de E et on veut lui associer une classe.

Comme problème jouet, on considère donc que $E = \mathbb{R}^2$ et que l'on prend la distance euclidienne.

Principe de l'algorithme



L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.

L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.

L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.

L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.

L'algorithme des KNN consiste à :

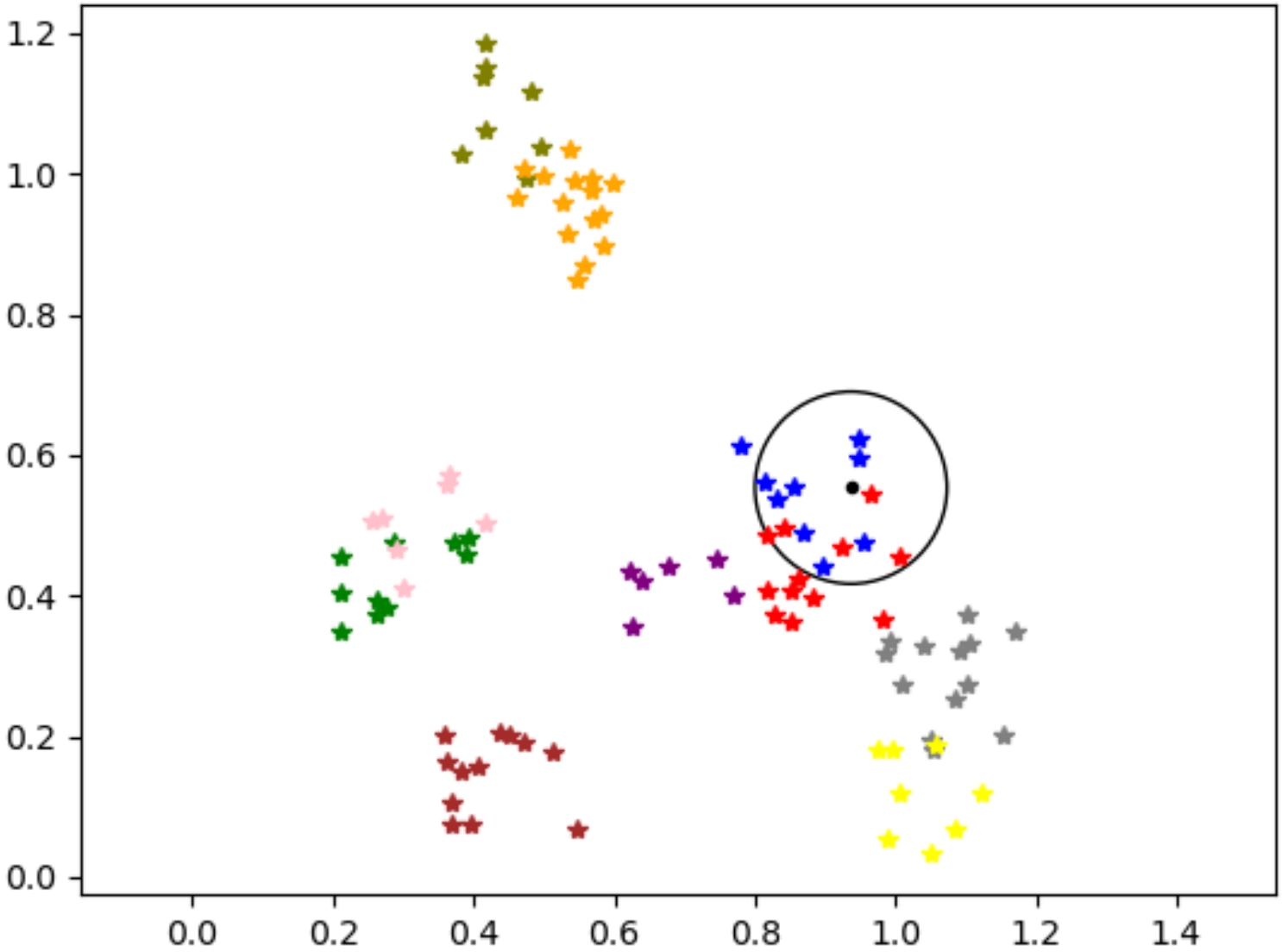
- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.

L'algorithme des KNN consiste à :

- mesurer les distance $d(y, x_j)$ pour $j \in [\![1, N]\!]$,
- déterminer les K plus proches voisins de y. Les K éléments les plus proches de y sont notés : $x_{i1}, x_{i2}, \ldots, x_{iK}$.
- Choisir pour classe de y la classe la plus fréquente parmi celles des plus proches voisins : $(x_{i1}, x_{i2}, \ldots, x_{iK})$.

- Le choix du paramètre K (nombre de voisins que l'on prend en compte) est critique. Il n'est pas facile de régler ce paramètre et de l'interpréter.
- On peut avoir plusieurs classes avec l'effectif maximal parmi les K voisins. Dans ce cas, on choisit arbitrairement une classe au nouvel objet y parmi les plus fréquentes.


1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.

```
def distance(y, lX) :
"""
entrée:
lX = liste d'objets (éléments de R**2) de longueur N
y = objet (élément de R**2)
on dispose d'une fonction d
qui mesure la distance entre objets
sortie:
lD = liste de float
   = liste des  distances d(y, xi) pour i dans [|1 , N|]
"""
```

- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.

```
def KNN(D, K):
    """
    entrée: D = liste de float
              = liste des distances entre l'objet à placer
                y et chacun des objets
                de la liste d'apprentissages
            K = int
              = nombre de voisins à prendre en compte
    sortie: IKNN = liste de int de longueur K
                 = liste des indices
                       des K plus proches voisins.
    """
```

- Déterminer la classe la plus fréquente parmis les voisins

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

Les étapes du programme

- Données : `lX` liste d'objets de longueur `N`, `y` un objet, `lC` la liste des classe de chaque objet et une fonction `d` pour la distance.
- Mesurer les distances ente l'objet y et chacun des objets x_j pour $j \in [\![1, N]\!]$.
- Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- Déterminer la classe la plus fréquente parmis les voisins.

```
def classe(lKNN, lC):
    """
    entrée:
    lKNN = liste de int de longueur K
         = liste des indices des K plus proches voisins.
    lC = liste de int
       = liste des classes
    sortie:
    cy =
    valeur de la classe la plus fréquente parmi les kNN
    """
```

Mesure des distances

```
def distance(y, IX) :
N = len(X)
ID = [0] *N
for k in range(N):
  ID[k] = d(y , IX[k])
return ID
```

On aurait aussi pu faire en une ligne :

```
ID = [ d(y , IX[k]) for k in range(N) ]
```

Rappel sur le tri rapide

```
def tri(L):
    """
    entrée: L = list= la liste que l'on souhaite trier.
    sortie: list= la liste triée
    """
    if len(L) <= 1 :
        return L
    pivot = L.pop()
    Lpetit = [ x for x in L if x <= pivot]
    Lgrand = [ x for x in L if x > pivot]
    return tri(Lpetit) + [pivot] + tri(Lgrand)
```

- Pas optimal avec cette programmation : on est obligé de recopier la liste plutôt que la trier sur place (mais plus simple à comprendre).
- Il est plus judicieux de chercher que les K plus petits éléments que de trier toute la liste.
- Voir les optimisations sur le poly.

Rappel sur le tri rapide

```
def tri(L):
    """
    entrée: L = list= la liste que l'on souhaite trier.
    sortie: list= la liste triée
    """
    if len(L) <= 1 :
        return L
    pivot = L.pop()
    Lpetit = [ x for x in L if x <= pivot]
    Lgrand = [ x for x in L if x > pivot]
    return tri(Lpetit) + [pivot] + tri(Lgrand)
```

- Pas optimal avec cette programmation : on est obligé de recopier la liste plutôt que la trier sur place (mais plus simple à comprendre).
- Il est plus judicieux de chercher que les K plus petits éléments que de trier toute la liste.
- Voir les optimisations sur le poly.

Rappel sur le tri rapide

```
def tri(L):
    """
    entrée: L = list= la liste que l'on souhaite trier.
    sortie: list= la liste triée
    """
    if len(L) <= 1 :
        return L
    pivot = L.pop()
    Lpetit = [ x for x in L if x <= pivot]
    Lgrand = [ x for x in L if x > pivot]
    return tri(Lpetit) + [pivot] + tri(Lgrand)
```

- Pas optimal avec cette programmation : on est obligé de recopier la liste plutôt que la trier sur place (mais plus simple à comprendre).
- Il est plus judicieux de chercher que les K plus petits éléments que de trier toute la liste.
- Voir les optimisations sur le poly.

Recherche des k plus proches voisins

- on veut trier la liste des distances en conservant les indices (car cela correspond aux classes dans `1C`)
- On crée donc une liste de couple [indice , distance] que l'on trie en prenant en compte uniquement la distance (deuxième élément).

Recherche des k plus proches voisins

- on veut trier la liste des distances en conservant les indices (car cela correspond aux classes dans `lC`)
- On crée donc une liste de couple [indice , distance] que l'on trie en prenant en compte uniquement la distance (deuxième élément).

Recherche des k plus proches voisins

- on veut trier la liste des distances en conservant les indices (car cela correspond aux classes dans `1C`)
- On crée donc une liste de couple [indice , distance] que l'on trie en prenant en compte uniquement la distance (deuxième élément).

Recherche des k plus proches voisins

- on veut trier la liste des distances en conservant les indices (car cela correspond aux classes dans `1C`)
- On crée donc une liste de couple [indice, distance] que l'on trie en prenant en compte uniquement la distance (deuxième élément).

```
def KNN(D, K):
    # on crée une liste distance indice,
    listeAtrier = [ [i, D[i] ] for i in range(len(D))]
    listeAtrier = triRapideInd(listeAtrier)
    # on extrait les k premiers indices
    LKNN = []
    for i in range(K):
        LKNN.append( listeAtrier[i][0])
    ## autre méthode:
    ## LKNN = [ listeAtrier[i][0] i in range(K) ]
    return LKNN
```

Recherche des k plus proches voisins

- On adapte l'algorithme de tri rapide au cas d'une liste de couple [indice , distance]

Recherche des k plus proches voisins

- On adapte l'algorithme de tri rapide au cas d'une liste de couple [indice , distance]

```
def triRapideInd(Li):
    """
    entrée: Li = liste du type indice, valeur
    sortie: Li triée en fonction de la valeur
    """
    if len(Li) <= 1 :
        return Li

    pivot = Li.pop()
    Lpetit = [ x  for x in Li if x[1] <= pivot[1] ]
    Lgrand = [ x for x in Li if x[1] > pivot[1]]
    return ( triRapideInd(Lpetit)
             + [pivot]
             + triRapideInd(Lgrand) )
```

Choisir la classe la plus fréquente

- On compte combien de fois est présente chaque classe, ie l'effectif de chaque classe. On utilise un compteur.

Choisir la classe la plus fréquente

- On compte combien de fois est présente chaque classe, ie l'effectif de chaque classe. On utilise un compteur.

```
def classe(IKNN, IC):
  """
  entrée:
  IKNN = liste de int de longueur K
       = liste des indices des K plus proches voisins.
  IC = liste de int
     = liste des classes
  sortie:
  cy =
  valeur de la classe la plus fréquente parmi les kNN
  """
  # déterminer les effectifs de chaque classe
  nbrClasse = max(IC)+1 # on peut aussi mettre en entrée
  compteur = [0]*nbrClasse
  for ind in IKNN :
    compteur[ IC[ind] ] += 1
```

Choisir la classe la plus fréquente

- un simple algorithme de recherche de maximum donne ensuite la classe la plus fréquente.

Choisir la classe la plus fréquente

- un simple algorithme de recherche de maximum donne ensuite la classe la plus fréquente.

```
def classe(lKNN, lC):
  # déterminer les effectifs de chaque classe
  nbrClasse = max(lC)+1 # on peut aussi mettre en entrée
  compteur = [0]*nbrClasse
  for ind in lKNN :
    compteur[ lC[ind] ] += 1
  # détermine la classe la plus fréquente
  cy = 0 # classe la plus fréquente
  emax = compteur[0] # effectif maximum
  for i in range(nbrClasse) :
    if compteur[i] > emax :
      cy = i
      emax = compteur[i]
  return cy
```

Code complet

```
D = listedesdistances(IX , y)

LKNN = KNN(D, 8)
print("voici les points les plus proches")
for ind in LKNN :
    print("point d'indice", ind,
          "\t coordonnées:", IX[ind][0], ", ", IX[ind][1],
          "\t classe:", IC[ind])

cy = classe(LKNN, IC)
print("classe du point y", cy,
          "il est colorié en ", LCOULEUR[cy])
```

Matrice de confustion

- La *matrice de confusion* permet de mesurer la qualité du système de classification.
- On prend M objets dont on connaît la classification (base de test). Cette classification est qualifiée de certaine. On applique l'algorithme à chacun de ses M objets et on note la classification obtenue (dite classification estimée).
- On met ces résultats dans une matrice : ligne i, colonne j on met le nombre d'éléments qui ont été classifiée dans la classe j alors qu'ils sont dans la classe i.
- Si la classification était parfaite, alors seule la diagonale aurait des éléments non nuls.
  Une classification est de qualité lorsque chaque ligne contient 95% de ses valeurs sur l'élément diagonal.

Matrice de confustion

- La *matrice de confusion* permet de mesurer la qualité du système de classification.
- On prend M objets dont on connaît la classification (base de test). Cette classification est qualifiée de certaine. On applique l'algorithme à chacun de ses M objets et on note la classification obtenue (dite classification estimée).
- On met ces résultats dans une matrice : ligne i, colonne j on met le nombre d'éléments qui ont été classifiée dans la classe j alors qu'ils sont dans la classe i.
- Si la classification était parfaite, alors seule la diagonale aurait des éléments non nuls.
  Une classification est de qualité lorsque chaque ligne contient 95% de ses valeurs sur l'élément diagonal.

Matrice de confustion

- La *matrice de confusion* permet de mesurer la qualité du système de classification.
- On prend M objets dont on connaît la classification (base de test). Cette classification est qualifiée de certaine. On applique l'algorithme à chacun de ses M objets et on note la classification obtenue (dite classification estimée).
- On met ces résultats dans une matrice : ligne i, colonne j on met le nombre d'éléments qui ont été classifiée dans la classe j alors qu'ils sont dans la classe i.
- Si la classification était parfaite, alors seule la diagonale aurait des éléments non nuls.
  Une classification est de qualité lorsque chaque ligne contient 95% de ses valeurs sur l'élément diagonal.

Matrice de confustion

- La *matrice de confusion* permet de mesurer la qualité du système de classification.
- On prend M objets dont on connaît la classification (base de test). Cette classification est qualifiée de certaine. On applique l'algorithme à chacun de ses M objets et on note la classification obtenue (dite classification estimée).
- On met ces résultats dans une matrice : ligne i, colonne j on met le nombre d'éléments qui ont été classifiée dans la classe j alors qu'ils sont dans la classe i.
- Si la classification était parfaite, alors seule la diagonale aurait des éléments non nuls.
  Une classification est de qualité lorsque chaque ligne contient 95% de ses valeurs sur l'élément diagonal.

Algortihme des Kmeans

- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.

Algortihme des Kmeans

- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.

Algortihme des Kmeans

- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.

Algortihme des Kmeans

- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.

Algortihme des Kmeans

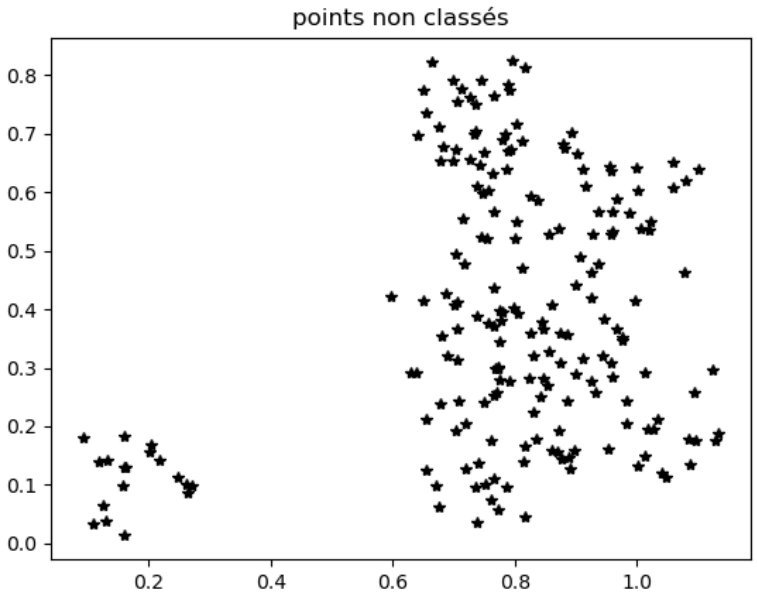
- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

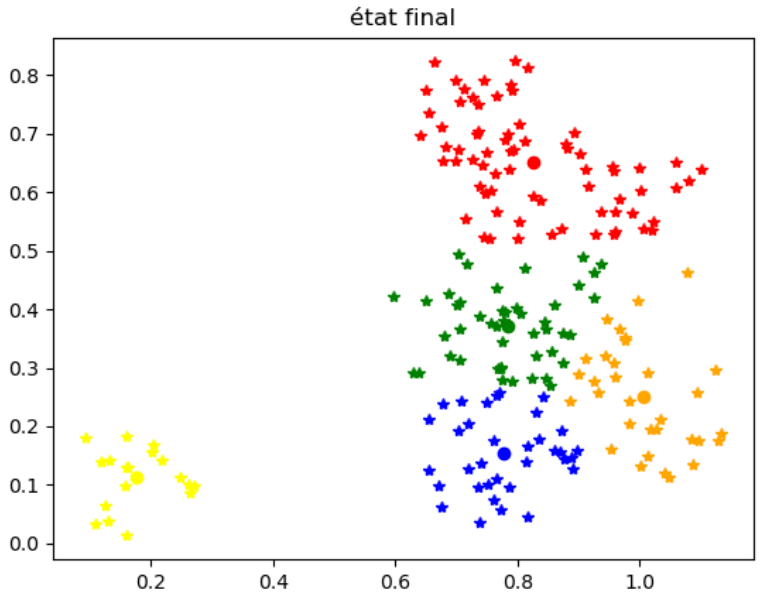
- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.

Algortihme des Kmeans

- Toujours le problème de la classification mais on ne suppose plus que les données sont déjà classifiées.
- On dispose ainsi de N objets que l'on voudrait répartir en k classes. La valeur de k est choisie par l'utilisateur.
- On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
- On parle aussi d'*apprentissage non supervisée* puisque l'algorithme apprend à créer des groupes sans partir de données (contrairement aux KNN).

- Pour notre problème jouet, on considérera toujours que l'on travaille avec des points de $\mathbb{R}^2$ et la distance euclidienne.


points non classés
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


état final
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0

Principe de l'algorithme des k-moyenne

- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

Principe de l'algorithme des k-moyenne

- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

Principe de l'algorithme des k-moyenne

- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

Principe de l'algorithme des k-moyenne

- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

Principe de l'algorithme des k-moyenne

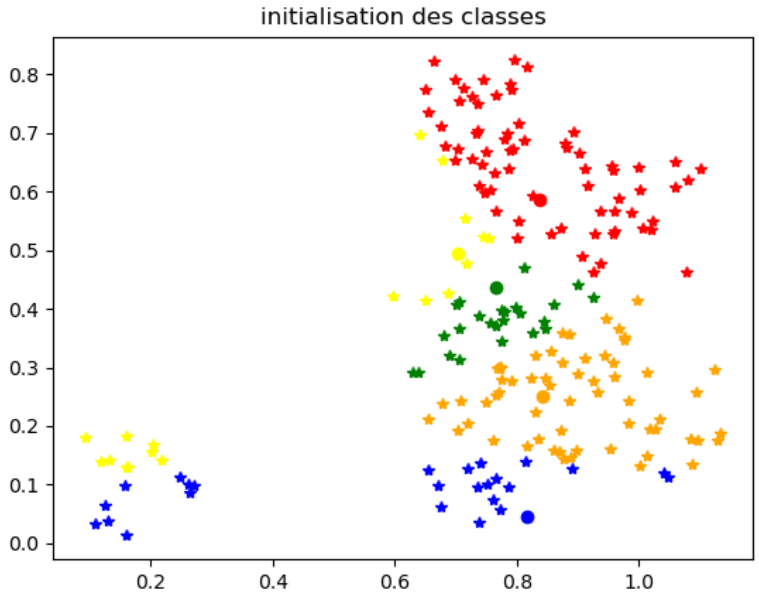
- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

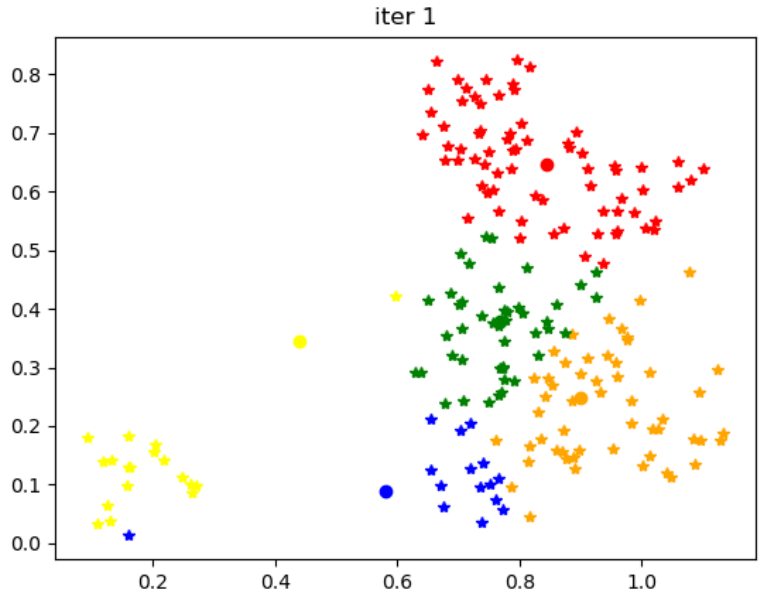
Principe de l'algorithme des k-moyenne

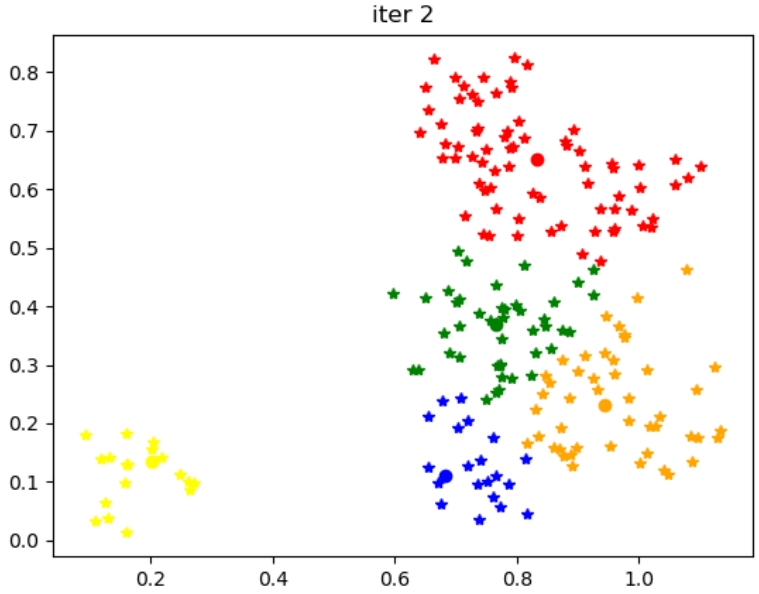
- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

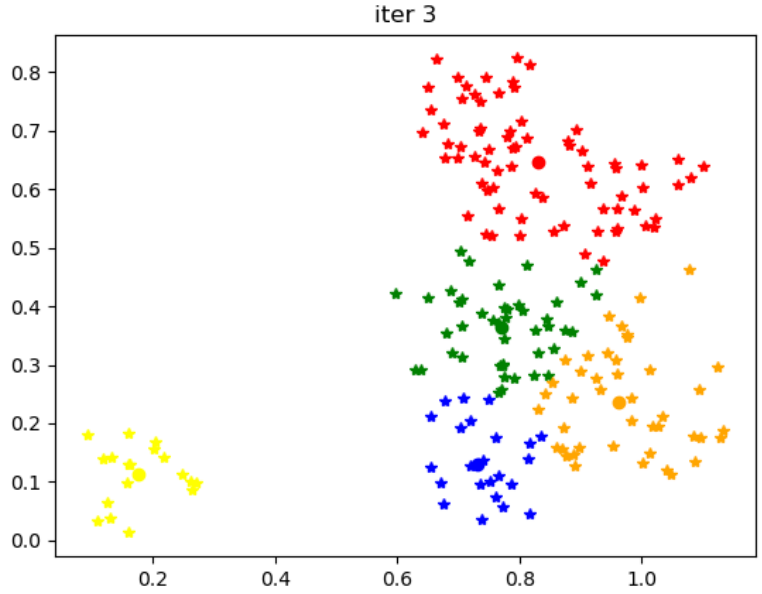
Principe de l'algorithme des k-moyenne

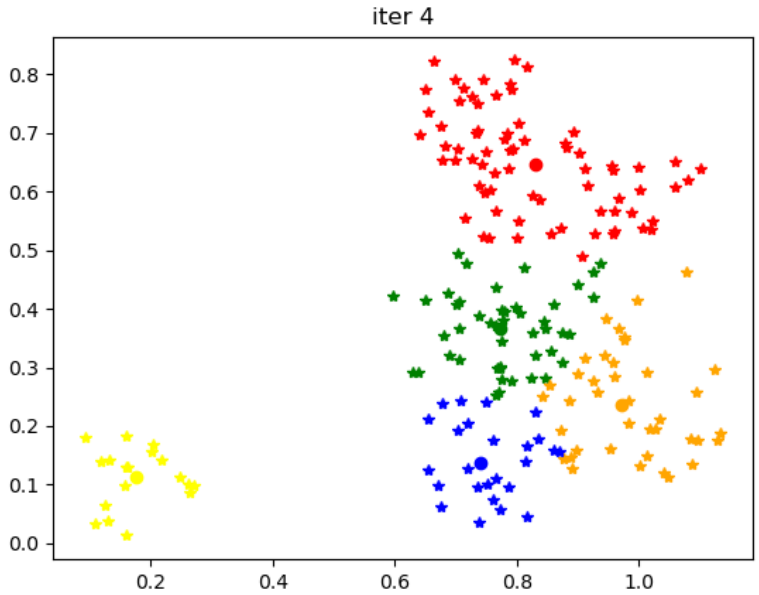
- L'algorithme des k-moyennes part d'une classification aléatoire et l'améliore au fur et à mesure.
- Chaque classe possède un CENTRE DE GRAVITÉ, qui est le barycentre des points qui font partie de cette classe.
- Le principe est d'initialiser arbitrairement les classes en donnant une valeur aux centres de gravité.
- Puis on attribue à chaque point la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité.
- Puis on attribue de nouveau à chaque point la classe dont le centre de gravité est le plus proche.
- On recommence ainsi, jusqu'à convergence ou dépassement d'un nombre d'itération fixé.

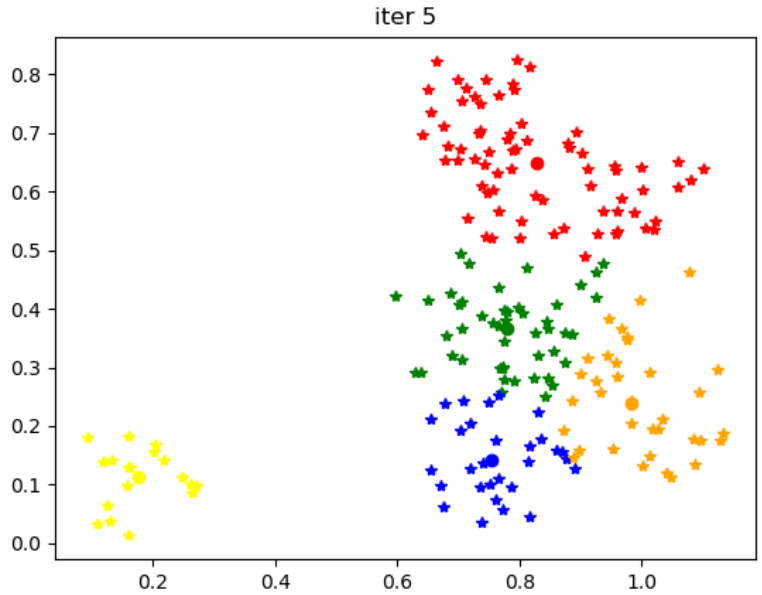

initialisation des classes
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0

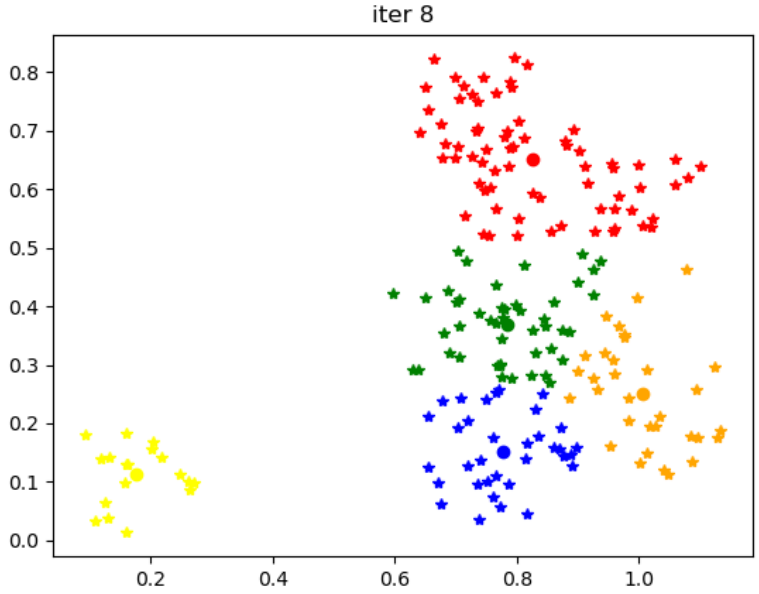

iter 1
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0

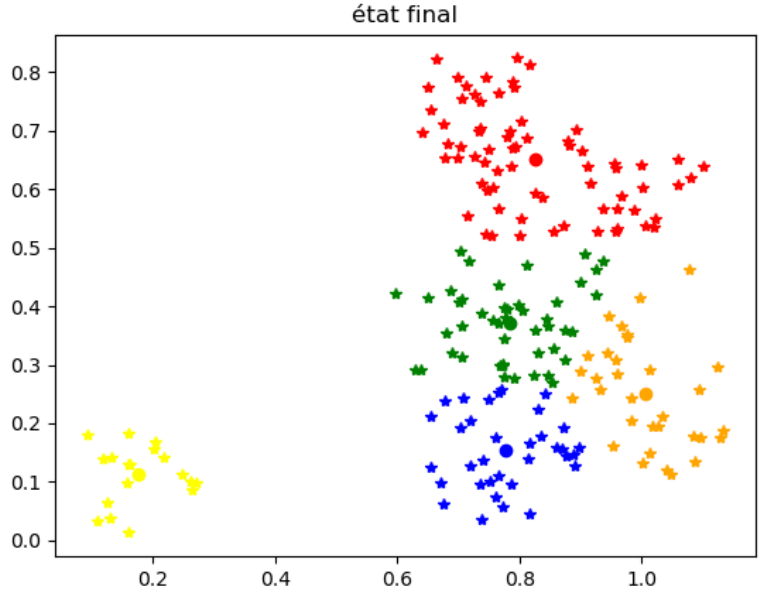

iter 2
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


iter 3
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


iter 4
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


iter 5
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


iter 8
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0


état final
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0

Limitation de l'algorithme

- L'ensemble qui contient les objets doit être un espace vectoriel (il faut pouvoir faire des moyennes). Il est muni d'une norme (que l'on choisit arbitrairement).
- La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
- De même, le choix de l'initialisation des classes est critique.
- Il est a priori possible que l'algorithme des K-means ne converge pas. C'est une *heuristique*.
- Que faire si une classe disparaît ?

Limitation de l'algorithme

- L'ensemble qui contient les objets doit être un espace vectoriel (il faut pouvoir faire des moyennes). Il est muni d'une norme (que l'on choisit arbitrairement).
- La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
- De même, le choix de l'initialisation des classes est critique.
- Il est a priori possible que l'algorithme des K-means ne converge pas. C'est une *heuristique*.
- Que faire si une classe disparaît ?

Limitation de l'algorithme

- L'ensemble qui contient les objets doit être un espace vectoriel (il faut pouvoir faire des moyennes). Il est muni d'une norme (que l'on choisit arbitrairement).
- La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
- De même, le choix de l'initialisation des classes est critique.
- Il est a priori possible que l'algorithme des K-means ne converge pas. C'est une *heuristique*.
- Que faire si une classe disparaît ?

Limitation de l'algorithme

- L'ensemble qui contient les objets doit être un espace vectoriel (il faut pouvoir faire des moyennes). Il est muni d'une norme (que l'on choisit arbitrairement).
- La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
- De même, le choix de l'initialisation des classes est critique.
- Il est a priori possible que l'algorithme des K-means ne converge pas. C'est une *heuristique*.
- Que faire si une classe disparaît ?

Limitation de l'algorithme

- L'ensemble qui contient les objets doit être un espace vectoriel (il faut pouvoir faire des moyennes). Il est muni d'une norme (que l'on choisit arbitrairement).
- La valeur de K est un paramètre critique de l'algorithme. Il n'y a aucune manière de le choisir automatiquement.
- De même, le choix de l'initialisation des classes est critique.
- Il est a priori possible que l'algorithme des K-means ne converge pas. C'est une *heuristique*.
- Que faire si une classe disparaît ?

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$. On peut par exemple poser $G_j = X_j$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
  Pour cela :
  - on calcule les distances $D_{i,j} = d(X_i, G_j)$ pour toutes les valeurs de j,
  - on détermine alors le centre de gravité le plus proche, ie la valeur de j_0 tel que :
    $$D_{i,j_0} = \min_{j\in[\![0,K-1]\!]} D_{i,j}$$
  - on associe à X_i la classe j_0 (ie la classe dont le centre de gravité est le plus proche)
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence.

Étude théorique de l'algorithme

- On considère donc N points de $\mathbb{R}^2$ notés $(X_i)_{i\in[\![0,N-1]\!]}$ que l'on veut classifier en K classes.
- On initialise les centres de gravité notés $(G_j)_{j\in[\![0,K-1]\!]}$.
- Pour chaque point, on va lui attribuer la classe dont le centre de gravité est le plus proche.
- On remet alors à jour les centres de gravité, en donnant pour valeur à G_j le barycentre des points de classe j.
- On remet alors à jour la classification des points.
- On recommence les deux étapes précédentes jusqu'à convergence. Pour savoir si l'algorithme converge, on note si l'un des points a changé de classe ou on s'arrête à un nombre fixé d'itérations.

Mise en place de l'algorithme

- Prototype de la fonction :
- Initialisation des centres de gravité

```
ICG = [ [IX[i][0], IX[i][1] ] for i in range(K)]
```

- Initialisation des classes :

```
# les premières valeurs sont déjà faites
IC = [i for i in range(K)] + [None for i in range(K,n) ]
# initialisation des classes des points restants:
for i in range(K, n):
    # on calcule toutes les distances entre le point
    # i et les centres de gravité
    ID = [dist( ICG[j], IX[i] ) for j in range(K)]
    # on recherche le min
    ci = indMin(ID)
    # c'est cette classe que l'on donne au point i
    IC[i] = ci
```

Mise en place de l'algorithme

- Prototype de la fonction :

```
def KMeans(IX, K, nbrIterMax):
    """
    entrée: IX = liste de couple de float de longueur n
               = liste des objets
            K  = int
               = nbr de classes demandées.
            nbrIterMax = int
                       = nbr d'iter maximal
    sortie: ICG = liste de couple de float de longueur k
          = liste des coordonnées des centres de gravité
            IC = liste de int de longueur n
               = liste des classes de chacun des points.
    """
```

- Initialisation des centres de gravité

```
ICG = [ [IX[i][0], IX[i][1] ] for i in range(K)]
```

- Initialisation des classes :

```
# les premières valeurs sont déjà faites
```

Mise en place de l'algorithme

- Prototype de la fonction :
- Initialisation des centres de gravité

```
ICG = [ [IX[i][0], IX[i][1] ] for i in range(K)]
```

- Initialisation des classes :

```
# les premières valeurs sont déjà faites
IC = [i for i in range(K)] + [None for i in range(K,n) ]
# initialisation des classes des points restants:
for i in range(K, n):
    # on calcule toutes les distances entre le point
    # i et les centres de gravité
    ID = [dist( ICG[j], IX[i] ) for j in range(K)]
    # on recherche le min
    ci = indMin(ID)
    # c'est cette classe que l'on donne au point i
    IC[i] = ci
```

Mise en place de l'algorithme

- Prototype de la fonction :
- Initialisation des centres de gravité

```
ICG = [ [IX[i][0], IX[i][1] ] for i in range(K)]
```

- Initialisation des classes :

```
# les premières valeurs sont déjà faites
IC = [i for i in range(K)] + [None for i in range(K,n) ]
# initialisation des classes des points restants:
for i in range(K, n):
    # on calcule toutes les distances entre le point
    # i et les centres de gravité
    ID = [dist( ICG[j], IX[i] ) for j in range(K)]
    # on recherche le min
    ci = indMin(ID)
    # c'est cette classe que l'on donne au point i
    IC[i] = ci
```

Mise en place de l'algorithme

- La fonction indMin détermine l'indice du minimum est bien connue :
- Boucle principale : une variable booléenne permet de déterminer si il y a eu changement et un compteur du nombre d'itération.

```
nbrIter = 0
chgt = True # = False si pas de chgt durant un tour
while chgt and nbrIter < nbrIterMax :
    chgt = False
    nbrIter += 1
```

- La variable chgt est mise à False dès le début de la boucle et ne passe à True que si un point change de classe.

Mise en place de l'algorithme

- La fonction indMin détermine l'indice du minimum est bien connue :

```
def indMin(L):
    """
    entrée: L = list de float
    sortie indMin = indice du minimum
    """
    indMin = 0
    valMin = L[0]
    for i in range(len(L)):
        if L[i] < valMin:
            valMin = L[i]
            indMin = i
    return indMin
```

- Boucle principale : une variable booléenne permet de déterminer si il y a eu changement et un compteur du nombre d'itération.

```
nbrIter = 0
chgt = True # = False si pas de chgt durant un tour
while chgt and nbrIter < nbrIterMax :
    chgt = False
```

Mise en place de l'algorithme

- La fonction indMin détermine l'indice du minimum est bien connue :
- Boucle principale : une variable booléenne permet de déterminer si il y a eu changement et un compteur du nombre d'itération.

```
nbrIter = 0
chgt = True # = False si pas de chgt durant un tour
while chgt and nbrIter < nbrIterMax :
    chgt = False
    nbrIter += 1
```

- La variable chgt est mise à False dès le début de la boucle et ne passe à True que si un point change de classe.

Mise en place de l'algorithme

- La fonction indMin détermine l'indice du minimum est bien connue :
- Boucle principale : une variable booléenne permet de déterminer si il y a eu changement et un compteur du nombre d'itération.

```
nbrIter = 0
chgt = True # = False si pas de chgt durant un tour
while chgt and nbrIter < nbrIterMax :
    chgt = False
    nbrIter += 1
```

- La variable chgt est mise à False dès le début de la boucle et ne passe à True que si un point change de classe.

Mise à jour des centres de gravité

- Adaptation de l'algorithme des moyennes :

```
moyX = [0 for i in range(K)]
moyY = [0 for i in range(K)]
eff = [0 for i in range(K)] # effectif de chaque classe
for i in range(n):
    x,y = IX[i]
    c = IC[i]
    moyX[c] += x
    moyY[c] += y
    eff[c] += 1
ICG = [ [moyX[c] /eff[c], moyY[c] / eff[c] ]
                                    for c in range(K) ]
```

- On ne prend pas en compte ici le problème d'une classe qui devient vide !

Mise à jour des centres de gravité

- Adaptation de l'algorithme des moyennes :

```
moyX = [0 for i in range(K)]
moyY = [0 for i in range(K)]
eff = [0 for i in range(K)] # effectif de chaque classe
for i in range(n):
    x,y = IX[i]
    c = IC[i]
    moyX[c] += x
    moyY[c] += y
    eff[c] += 1
ICG = [ [moyX[c] /eff[c], moyY[c] / eff[c] ]
                                    for c in range(K) ]
```

- On ne prend pas en compte ici le problème d'une classe qui devient vide !

Mise à jour de la classification :

- Même technique que vue précédemment :

```
for i in range(n):
    # on calcule toutes les distances entre le point
    # i et les centres de gravité
    ID = [dist( ICG[j], IX[i] ) for j in range(K)]
    # on recherche le min
    ci = indMin(ID)
    # c'est cette classe que l'on donne au point i
    if IC[i] != ci :
        chgt = True
        IC[i] = ci
```

- On note si un point qui change de classe.

Mise à jour de la classification :

- Même technique que vue précédemment :

```
for i in range(n):
    # on calcule toutes les distances entre le point
    # i et les centres de gravité
    ID = [dist( ICG[j], IX[i] ) for j in range(K)]
    # on recherche le min
    ci = indMin(ID)
    # c'est cette classe que l'on donne au point i
    if IC[i] != ci :
        chgt = True
        IC[i] = ci
```

- On note si un point qui change de classe.

```
def KMeans(IX, K, nbrIterMax):
    n = len(IX)
    ICG = [ [IX[i][0], IX[i][1] ] for i in range(K)]
    IC = [i for i in range(K)] + [None for i in range(K,n) ]
    for i in range(K, n):
        ID = [dist( ICG[j], IX[i] ) for j in range(K)]
        ci = indMin(ID)
        IC[i] = ci

    nbrIter = 0
    chgt = True # = False si pas de chgt durant un tour
    while chgt and nbrIter < nbrIterMax :
        chgt = False
        nbrIter += 1

        moyX = [0 for i in range(K)]
        moyY = [0 for i in range(K)]
        eff = [0 for i in range(K)] # effectif de chaque classe
        for i in range(n):
            x,y = IX[i]
            c = IC[i]
            moyX[c] += x
            moyY[c] += y
            eff[c] += 1
        ICG = [ [moyX[c] /eff[c], moyY[c] / eff[c] ] for c in range(K) ]

        for i in range(n):
            ID = [dist( ICG[j], IX[i] ) for j in range(K)]
            ci = indMin(ID)
            if IC[i] != ci :
                chgt = True
                IC[i] = ci

    return IC, ICG
```