

Algorithme de classification

Informatique tronc commun

Sylvain Pelletier

PSI - LMSC

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.

Introduction

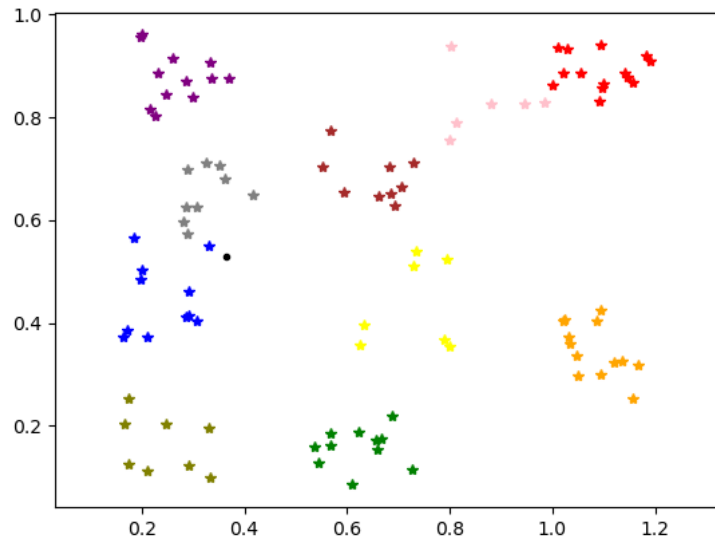
- Dans le programme, deux algorithmes pour l'intelligence artificielle : les k plus proches voisins (**KNN = k nearest neighbors**) et les k -moyennes (**k means**).
- Il s'agit de problème de **classification** : on dispose de mesures faites sur des objets et on veut les regrouper en groupes / classes.
- 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 ?

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 note $(x_j, c_j)_{j \in \llbracket 1, N \rrbracket}$. 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

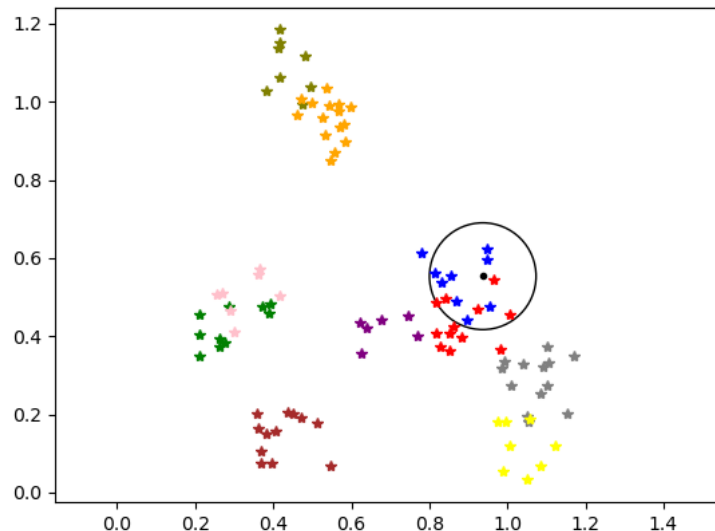


L'algorithme des KNN consiste à :

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

- ▶ 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.

Les étapes du programme



- ▶ Données : 1X liste d'objets de longueur N , y un objet, 1C 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 $j \in \llbracket 1, N \rrbracket$.
- ▶ Déterminer les k plus proches voisins, par un algorithme de tri rapide.
- ▶ Déterminer la classe la plus fréquente parmis les voisins.

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) ]
```

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 = triRapidelnd(listeAtrier)  
    # on extrait les k premiers indices  
    LKNN = []  
    for i in range(K):  
        LKNN.append( listeAtrier[i][0])  
    ## autre méthode:  
    ## LKNN = [ listeAtrier[i][0] for i in range(K) ]  
    return LKNN
```

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

```
def triRapidelnd(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 ( triRapidelnd(Lpetit)  
            + [pivot]  
            + triRapidelnd(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.

```
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.

```
def classe(IKNN, IC):
    # 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
    # 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])

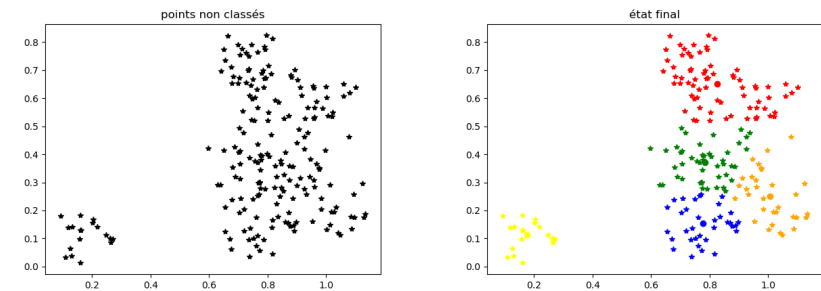
cy = classe(LKNN, IC)
print("classe du point y", cy,
      "il est colorié en ", LCOULEUR[cy])
```

Matrice de confusion

- 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 classifiés 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.

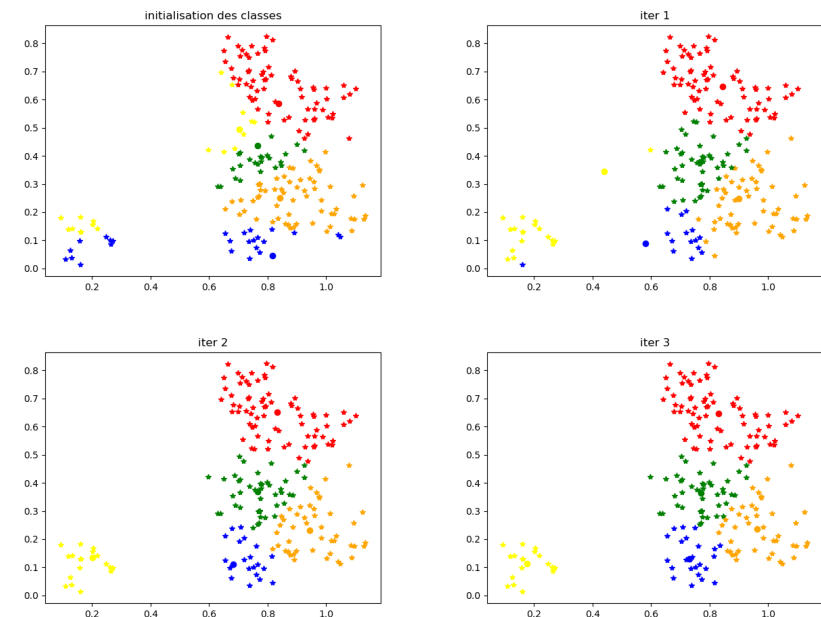
Algorithme des Kmeans

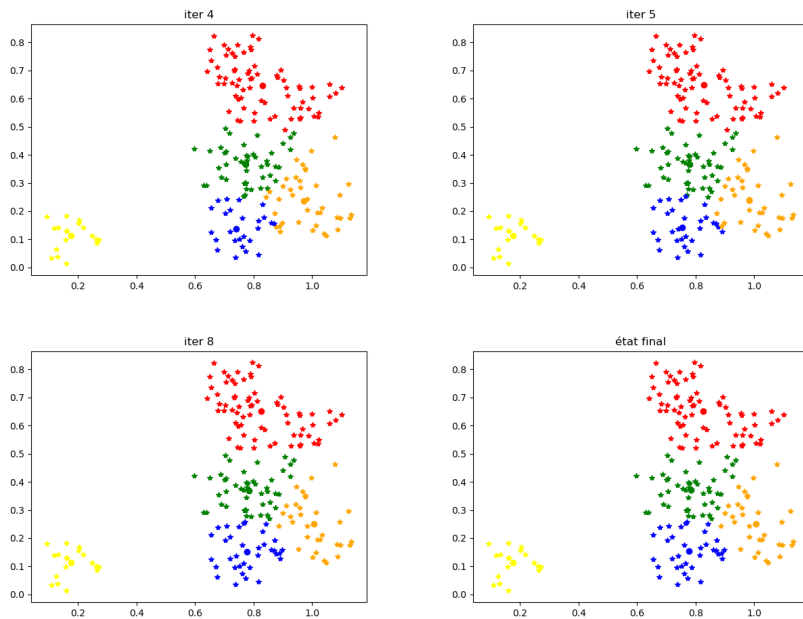
- ▶ Toujours le problème de la classification mais **on ne suppose plus que les données sont déjà classifiées**.
 - ▶ On dispose ainsi **de N objets que l'on voudrait répartir en k classes**. La valeur de k est choisie par l'utilisateur.
 - ▶ On parle de problème de *clustering* puisqu'il s'agit de créer des groupes.
 - ▶ On parle aussi d'*apprentissage non supervisé* 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.



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**.





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.

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 ?

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.

```
nbrlter = 0
chgt = True # = False si pas de chgt durant un tour
while chgt and nbrlter < nbrlterMax :
    chgt = False
    nbrlter += 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 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, nbrlterMax):
    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

    nbrlter = 0
    chgt = True # = False si pas de chgt durant un tour
    while chgt and nbrlter < nbrlterMax :
        chgt = False
        nbrlter += 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
```