Introduction aux méthodes d'intelligence artificielle

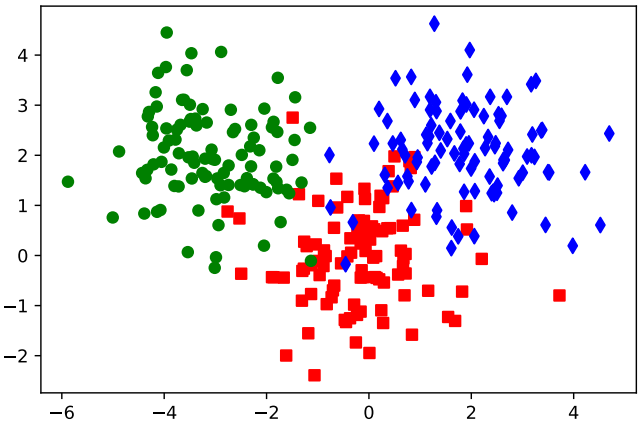
Introduction

Le terme intelligence artificielle est un terme général désignant un ensemble de méthodes informatiques destinées à imiter certaines capacités de l'intelligence humaine. Les domaines d'application sont nombreux : reconnaissance d'objets sur des images, programmation d'une machine prenant le rôle d'un joueur, classifications diverses ... Ces méthodes ont, parmi les méthodes informatiques, une particularité : elles impliquent souvent des algorithmes d'apprentissage. Cela signifie que le traitement de données massives par l'algorithme va permettre d'améliorer son efficacité.
Dans ce TP, nous allons traiter les deux méthodes suivantes : k plus proches voisins et k moyennes.

1 Méthode des k plus proches voisins

1.1 Description de la méthode

Cette méthode est destinée à traiter le problème suivant. On dispose d'une population (au sens large) divisée en plusieurs groupes. Les individus de cette population sont porteurs de caractéristiques numériques (âge, taille, intensité d'une couleur ...). L'objectif est, en considérant la valeur de ces caractéristiques pour un individu (dont le groupe est a priori inconnu), de le classer dans un des groupes.
Le fait que ces groupes sont connus à l'avance implique que cette méthode est classée parmi les méthodes d'apprentissage supervisé.
Pour pouvoir effectuer ce classement d'un élément dans un groupe en s'appuyant sur ses caractéristiques, on dispose au départ d'un échantillon de cette population appelé jeu de données d'apprentissage.
Par exemple, le graphique ci-dessous représente des points répartis en trois groupes : disques verts, losanges bleus, carrés rouges. Les caractéristiques de chaque point sont ses coordonnées. Peut-on, étant donné un point dont on ne connaît pas le groupe mais juste les coordonnées, décider du groupe auquel il appartient ?



Le principe de la méthode est le suivant. On fixe un entier k. Parmi le jeu de données d'apprentissage, on détermine les k points les plus proches du point auquel on doit affecter un groupe (au sens de la distance euclidienne, d'autres choix étant bien sûr possibles). On détermine ensuite le groupe majoritaire parmi ces k points et c'est ce groupe qui est affecté au point que l'on souhaite classer.
Nous allons construire cette méthode sur l'exemple simple de trois groupes de points dans le plan, comme sur le graphique ci-dessus. En pratique, cette méthode peut être appliquée à des problèmes plus complexes. On en verra un exemple (la reconnaissance de caractères) en TP.

1.2 Mise en place de la méthode

Le fichier `kNN_Acompleter.py` contient la trame du programme mettant en place la méthode.

1. La première partie du fichier est destinée récupérer un jeu de données d'apprentissage (les points dont le groupe est connu). Ce jeu a été créé pour le besoins du TP, ces données sont aléatoires ; mais dans les applications concrètes, il s'agit bien sûr de données provenant d'une partie de la population dont les caractéristiques et le groupe sont connus à l'avance.
On en profite pour revoir les commandes permettant de lire dans un fichier texte (ici le fichier 'donnees_app_KNN.txt').
Compléter le début de notre programme :

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


## RECUPERATION  DU JEU   DU JEU DE DONNEES D'APPRENTISSAGE
fichier = open('donnees_app_KNN.txt','r')
liste_lignes = fichier.readlines()
fichier.close()
```

La variable `liste_lignes` contient alors la liste des lignes contenues dans le fichier (ce sont des chaînes de caractères).
On aura besoin sur chacune de ces lignes :

- d'enlever le ' n' final grâce à la commande `ligne.strip()` ;
- de découper cette chaîne au niveau de chaque virgule grâce à la commande `ligne.split(',')` .

```
donnees_apprentissage = []
for ligne in liste_lignes:
    ligne =           #pour enlever le '\n' final
    x , y , couleur  =
    donnees_apprentissage.append( ((........ , ........), ........ ))
```

2. Écrire tout d'abord une fonction `dist(P1,P2)` qui renvoie la distance entre les points `P1` et `P2`.

```
def dist(P1,P2):
    """renvoie la distance entre deux points,
    représentés sous forme d'une liste
    ou d'un couple de deux éléments"""
    ...
    ...
    ...
```

Écrire ensuite une fonction `kNN(P,donnees,k)` qui renvoie les k plus proches voisins du point P dans la liste de points `donnees`.
Suggestion technique : on peut trier une liste d'objets (de couples par exemple) selon un critère sur ces objets de la manière suivante. On définit une fonction sur les objets contenues dans la liste, puis on passe cette fonction en paramètre de la fonction `sorted` de python. Tester par exemple :

```
def somme(C):
    return C[0]+C[1]

l = [ (0,1) , (4,5) , (1,3) , (2,1)]
ls = sorted(l , key = somme)
print(ls)
```

Remarque : du point de vue de la complexité, ce tri n'est pas un choix très pertinent. En effet, on pourrait déterminer le minimum des distances, puis le second minimum, ..., jusqu'au k-ème. Si k est petit devant le nombre de points, cela est plus efficace.

Compléter :

```
def kNN(P,donnees,k):
    """renvoie les k plus proches voisins du point P
    dans la liste de points donnees ;
    la liste donnees est une liste de couples dont :
            - le premier termes est un couple représentant
                    les coordonnées d'un point,
            - le deuxième terme est la catégorie de ce point
    """
    #on définit tout d'abord une fonction sur les couples de donnees
    # qui va servir de critère de tri
    def distance_a_P(d):
        """ d est de la forme ((X,Y),'couleur')
        """
        return  ......

    #On trie les données selon cette fonction de comparaison
    donneesTriees = ....
    return .....
```

3. Il faut ensuite trouver le groupe (la couleur pour nous) la plus fréquente dans ces k plus proches voisins. On écrit pour cela une fonction `typekNN(P,donnees,k)` qui renverra une liste de couleurs, le fait que cette liste soit de longueur strictement supérieure à 1 signalant qu'il y a des ex aequo. Dans ce cas, on dira que la méthode échoue à décider du groupe.
   Suggestion technique : comme il n'y a que trois groupes, on pourrait calculer les effectifs de chacun et trouver le groupe d'effectif maximal de manière artisanale. On propose la méthode suivante, qui permettra de réemployer la structure de dictionnaire abordée au TP précédent. On utilise un dictionnaire avec les clefs 'red', 'blue', 'green' et les valeurs correspondant qui comptent le nombre d'apparitions de chaque couleur. Pour obtenir ces nombres, on parcourt la liste renvoyée par la fonction `kNN(P,donnees,k)`.
   On pourra écrire au préalable une fonction `argMax(dico)` qui renvoie la liste des clefs dont les valeurs associées sont maximales.
   Compléter :

```
def argMax(dico):
    """renvoie la liste des clefs dont les valeurs associées sont max"""
    M = - np.inf
    for valeur in dico....      :
        if ...       :
            M = ....
    l = []
    for a in ...          :
        if ...            :
            l.append(a)
    return l

def typekNN(P,donnees,k):
    """renvoie le type (parmi les trois couleurs) du point P
    détecté selon la méthode des k plus proche voisins,
    appliquée avec comme jeu de données la liste donnees.
    En cas d'égalité, la fonction renvoie une liste de plusieurs éléments"""
    kProchesVoisins = ...
    nombreCouleur = {'red' : 0 , 'blue' : 0 , 'green' : 0}
    # ce dictionnaire  devra contenir le nombre de fois où la couleur
    # 'couleur'  apparait dans la liste  des k plus proches voisins de P
    ...
    ...
    ...
    ...
    return couleurMax
```

1.3 Matrice de confusion

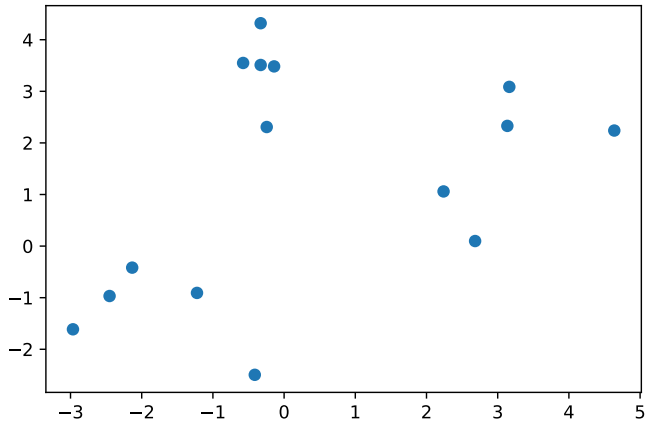
Les fonctions écrites ci-dessus sont destinées à affecter une couleur à un point dont on connaît uniquement les coordonnées. Cependant, avant d'utiliser cette méthode dans des cas concrets, on effectue en général une série de tests. Pour cela, il faut disposer d'un autre jeu de données, dont on connaît les coordonnées et le groupe. On donne ces données en exemple à la méthode et on regarde si pour chacune le groupe renvoyé est le bon.
Ces informations sont rassemblées dans un tableau appelé matrice de confusion : la case (i,j) de cette matrice contient le nombre d'éléments du jeu de test faisant partie du groupe i et que la méthode à affectés au groupe j. Bien sûr, si la méthode fonctionne parfaitement, cette matrice sera diagonale.
Construire la matrice de confusion associée à ce jeu de données test.

```
# RECUPERATION  DU JEU DE DONNEES TEST

fichier = open('donnees_test_KNN.txt','r')
liste_lignes = fichier.readlines()
fichier.close()

donnees_test = []
...
...
...
...


##OBTENTION DE LA MATRICE DE CONFUSION

#dictionnaire faisant correspondre un numéro à chaque couleur :
dC = {'red' : 0 , 'blue' : 1 , 'green' : 2}

# On choisit une valeur de k :
k = 5

#initialisation de la matrice de confusion :
matConf = np.zeros((3,3))
#remplissage de la matrice de confusion
for d in donnees_test:
    ...
    ...
    ...
    ...

print(matConf)
```

2 Algorithme des k-moyennes

Cet algorithme a également pour objectif de classer des éléments d'un ensemble dans des sous-ensembles, mais ces sous-ensembles ne sont pas connus à l'avance. Pour cette raison, cette méthode d'apprentissage est dite non-supervisée. En pratique, il s'agit pour la machine de regrouper des données proches en classes.

Considérons un exemple dans une situation simple. On dispose d'un ensemble de points que l'on souhaite classer en k sous-ensembles (la valeur de k est fixée par l'utilisateur).



L'ojectif de notre algorithme va être de classer ces points dans k ensembles $C_1, \ldots, C_k$ en minimisant la somme des moments d'inertie, c'est à dire, en notant b_i le barycentre de C_i, la somme des valeurs suivantes :

$$\sum_{p \in C_i} ||p - b_i||^2 \ .$$

Ce problème est a priori très couteux en temps.
Nous allons adopter la stratégie suivante :

- On choisit arbitrairement k points $x_1, \ldots, x_k$.
- On regroupe les points en classes C_i : la classe C_i est l'ensemble des points p tels que $||p - x_i||$ est minimal parmi les $||p - x_j||$.
- On recalcule les points $x_1, \ldots, x_k$ de la manière suivante : x_i est le barycentre des points de C_i.
- On réitère le procédé (regroupement-nouvelle liste des barycentres) jusqu'à ce que la liste des classes (et donc celle des barycentres) se stabilise.

Remarques :

- Le choix initial des points destinés à être les barycentres peut ne pas être tout à fait arbitraire.
- Nous admettrons que cette méthode converge (c'est à dire que l'algorithme s'arrête).
- Nous admettrons également que la configuration obtenue après arrêt est un minimum local pour la somme des moments d'inertie (on n'a pas forcément trouvé la solution optimale).

Nous programmerons cet algorithme en TP sur l'exemple ci-dessus.