Algorithme des K plus proches voisins (KNN)

L'algorithme K-NN (K-nearest neighbors) est une méthode d'apprentissage supervisé. Il peut être utilisé aussi bien pour la régression que pour la classification. Son fonctionnement peut être assimilé à l'analogie suivante "dis moi qui sont tes voisins, je te dirais qui tu es...".

Méthode de prédiction

Pour effectuer une prédiction, l'algorithme K-NN va se baser sur le jeu de données en entier. En effet, pour une observation, qui ne fait pas parti du jeu de données, qu'on souhaite prédire, l'algorithme va chercher les K instances du jeu de données les plus proches de notre observation. Ensuite pour ces K voisins, l'algorithme se basera sur leurs variables de sortie (output variable), pour calculer la valeur de la variable de l'observation qu'on souhaite prédire.

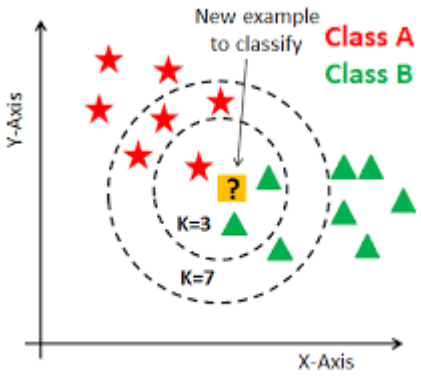

Figure 1: Méthode de prédiction

Programme Python

```python
def distance(p1,p2):
    '''
    Aide de la fonction distance du module knn :

    distance(p1,p2)
        Calcule la distance Euclidienne separant deux points p1 et p2

    ->Parametres : p1 : coordonnes de p1 (liste d'entiers)
                   p2 : coordonnes de p2 (liste d'entiers)

    -> Retour : distance separant p1 et p2 (float)
    Exemple : distance([0,1] , [1,1]) renvoie 1.0
    '''
    somme=0
    for i in range(len(p1)):
        somme+=(p1[i]-p2[i])**2
    return sqrt(somme)

def distance_voisins(p1,jeu2donnees):
    '''
    Aide de la fonction distance_voisins du module knn :

    distance(p1,dataset)
        Calcule la distance Euclidienne separant le point p1 et des diffrents points du jeu de donnees (dataset)

    ->Parametres : p1 : coordonnes de p1 (liste d'entiers)
                   jeu2donnees : liste de tuples : [('nom du joueur','poste du joueur','taille','masse')....]

    -> Retour : liste de listes contenant l'les caractéristiques du joueur et la distance entre ce joueur et le joueur p1
    Exemple : distance_voisins([180,90] , [('joueur1','arriere','175','75'), ('joueur2,'avant','190','90')]) renvoie
    [[('joueur1','arriere','175','75'), 15.811388300841896], [('joueur2,'avant','190','90'), 10.0]]
    '''
    result = []
    i=0
    for i in range(len(jeu2donnees)):
        result.append([jeu2donnees[i],distance(p1,[int(jeu2donnees[i][2]),int(jeu2donnees[i][3])])])
    return result
```

```python

def k_voisins(distances,k) :
    '''
    Aide la fonction K_voisins du module Knn :

    k_voisins(jeu2donnees,k)
        Renvoie la liste des K plus proches voisins à partir de la liste des distances
    Parametres : distances : liste de listes contenant les caracteristiques du joueur et la distance entre ce joueur et le joue
                 K : nombre de voisins considérés (int)
    Retour : Liste des K plus proches joueurs
    '''
    k_voisins = []
    for i in range(k) :
        dmin = distances[0][1]
        index = 0
        for i in range(len(distances)) :
            if distances[i][1] < dmin :
                dmin = distances[i][1]
                index = i
        k_voisins.append(distances.pop(index))
    return k_voisins

def predire_classe(k_voisins):
    '''
    Aide de la fonction predire_classe du module knn :

    predire_classe(proches joueurs)
        Renvoie la classe plus proche voisin du point p1

    ->Parametres : k_voisins : liste des k voisins

    -> Retour : Poste (classe) du plus proche voisin de p1
    '''
    postes = {}
    #Creer dictionnaire avec classes et nombre d'occurrence de chaque classe
    for joueur in k_voisins :
        if joueur[0][1] in postes :
            postes[joueur[0][1]] +=1
        else :
            postes[joueur[0][1]] = 1
    #Cherhce maximum du nombre d'occurence
    n = 0
    for classe in postes :
        if postes[classe] > n :
             n = postes[classe]
             classe_majoritaire = classe
    return classe_majoritaire



dataset = extractionDonnees("joueursToulouse.csv")
distances = distance_voisins([185,114],dataset)
kNN = k_voisins(distances,10)
prediction = predire_classe(kNN)
```

L'optimisation en informatique

L'optimisation est un procédé consistant à déterminer les paramètres d'un problème amenant à maximiser ou minimiser une fonction. Lorsque la fonction à optimiser est une fonction analytique clairement définie, des approches mathématiques, analytiques (étude des dérivées successives de la fonction, etc...) sont envisageables. Par contre lorsque que la fonction est plus difficilement calculable des approches numériques sont exploitées. Il existe pléthore de méthodes numériques adaptées à différent types de problèmes.
L'objectif du cours n'est pas de faire de vous des spécialistes des méthodes d'optimisation, mais de présenter quelques problèmes d'optimisation et leurs méthodes de résolution en présentant les avantages et les limites de celle-ci.

1. La méthodes gloutonnes

Principe de la méthode

Un algorithme glouton (on dit aussi gourmand, greedy en anglais) est un algorithme simple et intuitif utilisé dans les problèmes où l'on recherche une solution optimale.
La résolution du problème d'optimisation se fait par une décomposition en sous-problèmes où à chaque étape, l'algorithme Glouton fait un choix optimal avec l'idée de trouver la solution optimale pour résoudre le problème dans son ensemble.

Attention, ce principe n'assure absolument pas de trouver la «vraie» solution optimale du problème mais fournit une solution généralement intéressante et valide.

Problèmes résolus efficacement par la méthode gloutonne

✗ Problème du rendu de monnaie , Problème du sac à dos, problème du plus court chemin résolu par l'algorithme de Dijkstra

2. La méthode « diviser pour régner »

La méthode « diviser pour régner » (divide-and-conquer en Anglais) consiste à découper un problème en sous-problèmes similaires (d'où l'algorithme récursif résultant) afin de réduire la difficulté du problème initial. On espère casser récursivement le problème en sous-problèmes de plus en plus petits jusqu'à obtenir des cas simples permettant une résolution directe.
La méthode de "Diviser pour régner" en algorithmique se décompose en trois étapes :

➔ Diviser : on divise l'instance de départ en de plus petites.
➔ Régner : on résout l'algorithme sur les "petites" instances précédentes.
➔ On fusionne les résultats précédents pour obtenir le résultat final.

Problèmes résolus efficacement par la méthode diviser pour régner

✗ Tri fusion, Tri rapide (quicksort), rotation d'image sans copie

Le principe "diviser pour régner" permet d'écrire des algorithmes avec une complexité qui peut être plus faible que d'autres algorithmes pour un même problème (Théorème maître).