**Algorithme des $k$-plus proches voisins**

Étant donnés dans $\mathbb{R}^d$ un ensemble $E = \{x_1, ..., x_n\}$ de points dont on connaît une partition de $E$ en $p$ classes $A_1, ..., A_p$, on souhaite attribuer une classe à un nouvel élément $x$ en lui attribuant la classe majoritaire parmi les $k$-plus proches voisins.

## Code Python

Un point de $\mathbb{R}^d$ est codé par une liste.

On considère un ensemble $E$ de $n$ points de $\mathbb{R}^d$ (codés par une liste de points indexée de 0 à $n-1$).

On code la partition par un tableau $A$ de longueur $n$ à valeurs dans $\{0, 1, ..., k-1\}$.

Ainsi, la classe d'indice $j$ est l'ensemble des éléments $E[i]$ tels que $A[i] = j$.

```
def distance(x,y) :
     d = len(x) ; s = 0
     for i in range(d) :
          s = s + (x[i]-y[i])**2
    return s**0.5

def plusProches(E,x,k) :
     # renvoie la liste L des indices i des k-plus proches voisins de x dans E
     # on classe dans L les éléments y = E[i] par valeur décroissante de d(x,y)
     # pour des raisons pratiques, on stocke les couples (i,δ), où δ = d(x,y)
     n = len(E)
     # on initialise L aux k premiers éléments de E, on utilise le tri par insertions
     L = [ (0,distance(x,E[0])) ]
     for i in range(1,k) :      # tri des k premiers termes
          d = distance(x,E[k]) ; L.append( (k,d) ) ; j = i
          while j > 0 and L[j-1][1] > d :
               L[j-1],L[j] = L[j],L[j-1]
     #  on passe en revue les autres éléments de E en modifiant L
     #  si on trouve des éléments plus proches que le dernier élément de L
     for i in range(k,n) :
          d = distance(x,E[i])
          if d < E[k-1][1] :
               L[k-1] = (i,d) ; j = k-1
```

```
            while j > 0 and L[j-1][1] > d :
                  L[j-1],L[j] = L[j],L[j-1]     # cf tri par insertions
      return L

def classeMaj(L,A)
      # renvoie la classe majoritaire parmi les éléments de L
      # on utilise un tableau de comptage (mais il faut d'abord déterminer la valeur de p
      p = 0
      for x in range(L) :
            (i,d) = x ; p = max(p,A[i])
      comptage = [0]*(p+1)
      for x in range(L) :
           (i,d) = x ; comptage(A[i]) += 1
      # on cherche la classe i contenant le plus d'éléments
      i = 0
      for j in range(p) :
            if comptage[i] > comptage[j] :  i = j
      return i
```

*Remarque* : Une variante de l'algorithme consiste à attribuer un poids $p(y)$ à chacun des $k$-plus proches voisins $y$ choisis d'autant plus grand que sa distance au point $x$ est petite : il suffit alors de calculer pour chaque classe la somme des $p(y)$ pour les voisins $y$ appartenant à cette classe.

## Algorithme des $k$-moyennes

Étant donnés dans $\mathbb{R}^d$ un ensemble $E = \{x_1, ..., x_n\}$ de points et un entier $k$, on souhaite réaliser une partition de $E$ en $k$ classes $A_1, ..., A_k$, souvent appelés *clusters*, de façon à minimiser

$$J(A_1, ..., A_k) = \sum_{j=1}^{k} \sum_{x \in A_j} \left\| x - \mu_j \right\|^2$$

où $\left\| x - \mu_j \right\|$ est la distance d'un point $x \in A_j$ à la moyenne $\mu_j$ des points de sa classe $A_j$.

**1) Algorithme :**

- Choisir $k$ points $\mu_1, ..., \mu_k$ qui représentent les futures positions moyennes
- Répéter jusqu'à ce qu'il y ait convergence de la partition (ou convergence numérique) :
    - On considère pour chaque point $x_i$ le point parmi $\mu_1, ..., \mu_k$ dont il est le plus proche
      On obtient ainsi une partition $A_1, ..., A_k$ de $E$

- Pour $1 \leq i \leq k$, calculer la moyenne $\mu_i$ des points appartenant à $A_i$

*Remarque* : Il faut choisir au départ les $\mu_i$ de sorte que les $A_i$ soient non vides.
Il suffit de choisir pour $\mu_1, ..., \mu_k$ des points distincts de $E$.

**2) Notion d'inertie**

On pose

$$J = \sum_{j=1}^{k} \sum_{x \in A_j} \left\| x - \mu_j \right\|^2 \text{ appelé inertie (= somme des variances)}$$

*Prop* : L'inertie $J$ diminue lors de chacune des phases de l'algorithme

*dem* :

(i) En remplaçant $\mu_j$ par la moyenne des éléments $x$ de $A_j$, on diminue la valeur de $J$.
En effet, de façon générale, $E((Z-\mu)^2)$ est minimale lorsque $\mu = E(Z)$.

(ii) Lorsque $x \in A_j$ et qu'il existe $i \neq j$ tel que $\|x - \mu_i\| < \left\|x - \mu_j\right\|$, on diminue $J$ lorsqu'on fait passer $x$ dans la classe $A_i$.

*Remarque* : L'algorithme converge vers un minimum local de l'inertie : cette valeur dépend des choix initiaux. Parfois, on prend plusieurs valeurs initiales afin de retenir l'inertie minimale obtenue.

**Remarque : Dans l'algorithme des $k$-moyennes, la valeur de $k$ est fixée au départ.**

On peut se demander **comment choisir $k$ de façon optimale** (ni trop grand ni trop petit ...). Une méthode consiste à calculer l'inertie pour des valeurs de $k$ croissantes et de s'arrêter lorsque l'inertie cesse de diminuer notablement.

**3) Complément informatique : un autre algorithme de partitionnement**

*Algorithme* : On part de la partition en singletons et à chaque étape on fusionne deux classes en choisissant ceux pour lesquels l'augmentation de l'inertie est minimale.

En fusionnant deux classes d'indices $i$ et $j$, le terme

$$\sum_{x \in A_i} \|x - \mu_i\|^2 + \sum_{x \in A_j} \|x - \mu_i\|^2$$

va être remplacé par

$$\sum_{x \in A_i \cup A_j} \|x - \mu\|^2 \quad , \quad \text{où} \quad \mu = \frac{n_i \mu_i + n_j \mu_j}{n_i + n_j}$$

On a $\sum_{x \in A_i \cup A_j} \|x - \mu\|^2 = \sum_{x \in A_i} \|x - \mu_i\|^2 + \sum_{x \in A_j} \|\mu_i - \mu\|^2 = \sum_{x \in A_i} \|x - \mu_i\|^2 + n_i \|\mu_i - \mu\|^2$.
On a $n_i \|\mu_i - \mu\|^2 = \dfrac{n_i n_j}{n_i + n_j} \left\|\mu_i - \mu_j\right\|^2$.

Donc cette fusion induit une augmentation de l'inertie égale à $\dfrac{2 n_i n_j}{n_i + n_j} \left\|\mu_i - \mu_j\right\|^2$.

On choisit donc $i$ et $j$ (distincts) de sorte que $\dfrac{2 n_i n_j}{n_i + n_j} \left\|\mu_i - \mu_j\right\|^2$ soit minimale.

**4) Code Python de l'algorithme des $k$-moyennes**

Un point de $\mathbb{R}^d$ est codé par une liste.

On considère un ensemble $E$ de $n$ points de $\mathbb{R}^d$ (codés par une liste de points indexée de 0 à $n-1$).

On prend $k \leq n = \operatorname{card} E$ et on choisit les $k$ premiers points pour les valeurs initiales des $\mu_k$.

On code la partition par un tableau $A$ de longueur $n$ à valeurs dans $[\![k]\!] = \{0, 1, ..., k-1\}$.

Ainsi, la classe d'indice $j$ est l'ensemble des éléments $E[i]$ tels que $A[i] = j$.

```
def distance(v,w) :
     d = len(v) ; s = 0
     for i in range(d) :
          s = s + (v[i]-w[i])**2
     return s**0.5

def sommes(X,Y) :    ### sommation vectorielle de deux listes
     d = len(X) ; L = []
     for i in range(d) :
         L.append(X[i]+Y[i]]
     return L

def produit(lambda,X) :    ### produit d'un réel et d'une liste (codant un vecteur)
     d = len(X) ; L = []
     for i in range(d) :
         L.append(lambda*X[i])
     return L

def moyennes(E,A,k) :     # procédure qui renvoie le tableau L des k moyennes
     d = len(A[0]) ; n = len(E)
     L = [[0]**n for j in range(k)]
     c = [0]*k   # permet de compter le nombre d'éléments par classe
     for i in range(n) :
          j = A[i] ; L[j] = somme(L[j],E[i]) ; c[j] = c[j] + 1
     for j in range(k) :  L[j] = produit(c[j],L[j])
     return L

def plusProche(point,L) :
     # L est une liste de k points (représentant en fait les anciennes moyennes des classes)
     # on renvoie l'indice j tel que la distance entre L[j] et le point est minimale
     # s'il y a égalité entre plusieurs distances, on choisit le plus petit indice
     k = len(L) ; jMin = 0
     for j in range(1,k) :
          if distance(point,L[j])<distance(E[i],L[jMin]) :  jMin=j
     return jMin
```

```
def partition(E,L,k) :    # renvoie la partition associée à L
    n = len(E) ; A = []
    for i in range(n) :
        point = E[i]
        A.append(plusProche(point,L))
    return A

def algoKmoy(E,k,N) :
    # on suppose les points de E distincts et on effectue au plus N itérations
    n = len(E)
    L = [E[j] for j in range(k)] ; flag = false ; iter = 0
    while flag :
        B = A
        A = partition(E,L,k)
        iter = iter + 1 ; flag = (B != A) and iter < N
```