A K-moyennes en dimension 1

Les enseignants d'une CPGE scientifique souhaitent créer des groupes de niveau pour mieux adapter le contenu des TD. Ils disposent pour cela des notes des élèves et veulent les regrouper selon la similitude de leur moyenne générale. Dans ce but, ils utilisent l'algorithme des K-moyennes.

Formellement, les notes constituent un ensemble E de n réels $x_0 \leq x_1 \leq \leq x_{n-1}$. Dans le cadre de l'algorithme des K-moyennes, on cherche à déterminer une partition de $[\![0, n-1]\!]$ en $K \leq n$ sous-ensembles $\mathcal{P} = \{\mathcal{P}_0, \mathcal{P}_1, , \mathcal{P}_{K-1}\}$ non vides **tels que le score**

$$S(\mathcal{P}) = \sum_{i=0}^{K-1} \sum_{j \in \mathcal{P}_i} (x_j - \mu_i)^2 \tag{1}$$

soit minimal, où

$$\mu_i = \frac{1}{|\mathcal{P}_i|} \sum_{j \in \mathcal{P}_i} x_j \tag{2}$$

est la moyenne des éléments correspondant à la partie $\mathcal{P}_i$. Autrement dit, on veut minimiser la somme des carrés des écarts de chaque élément à la moyenne de sa partie.

■ **Exemple 1 — Exemple de solution optimale.** Par exemple, pour $E = \{1,2,3,5,8,10,14,15,18\}$, une solution optimale pour $K = 3$ est représentée sur la figure 1. Pour $K = 3$, on trouve $\mathcal{P} = \{\{0,1,2,3\},\{4,5\},\{6,7,8\}\}$ de $[\![0,8]\!]$. Cela signifie que les groupes de TD seront constitués comme suit :

- $\mathcal{P}_0$: les élèves de moyenne générale entre 0 et 5.
- $\mathcal{P}_1$: les élèves de moyenne générale entre 8 et 10.
- $\mathcal{P}_2$: les élèves de moyenne générale entre 14 et 18.

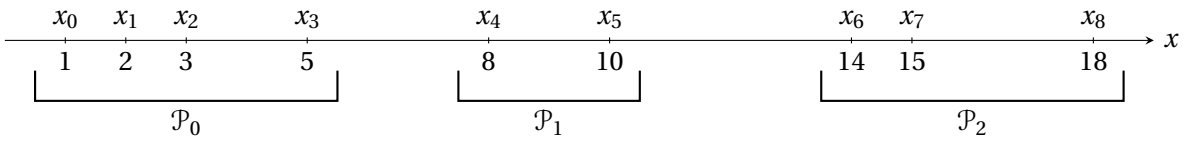


FIGURE 1 – Exemple de partition optimale pour l'ensemble $E = \{1,2,3,5,8,10,14,15,18\}$

R L'ensemble des données de cette partie est une liste de nombres. Une partition est représentée par une liste de liste.

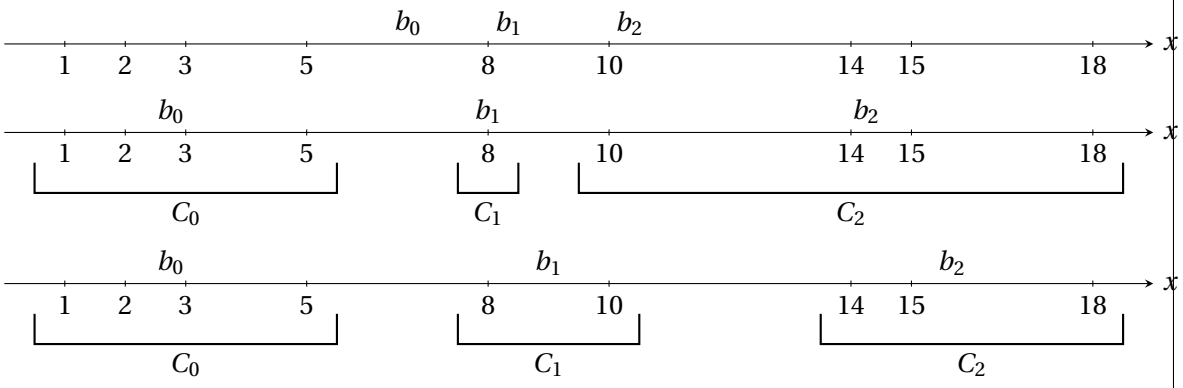
A1. Trouver la partition optimale au sens des K-moyennes pour $K = n$ et $K = n-1$. Justifier.

Solution : Si $K = n$, chaque partie est de cardinal 1. Ce qui est bien minimal, on ne peut construire qu'une seule partition de K parties.
Si $K = n-1$, toutes les parties sont de cardinal 1, sauf une qui est de cardinal 2. Pour que la partition soit optimale, il faut mettre dans le même groupe les deux éléments les plus proches dans E.

A2. Appliquer à la main [1] l'algorithme des K-moyennes sur l'ensemble E de l'exemple 1 pour $K = 3$. La partition iniale est `[[1, 5, 14], [2, 8, 15], [3, 10, 18]]`. Détailler les étapes de calculs jusqu'à convergence de l'algorithme.

Solution : L'algorithme se déroule comme suit :

```
[[1, 5, 14], [2, 8, 15], [3, 10, 18]] [6.67, 8.34, 10.34]
[[1, 2, 3, 5], [8], [10, 14, 15, 18]] [2.75, 8.0, 14.25]
[[1, 2, 3, 5], [8, 10], [14, 15, 18]] [2.75, 9.0, 15.67]
```



A3. Écrire une fonction de prototype `create_parts(E, k)` dont les paramètres sont la liste des éléments `E` et le nombre de parties `k`. Elle renvoie une liste de listes correspondant à une partition de `E`. Cette fonction parcourt tous les éléments de `E` et complète chaque partie en mode tourniquet : chaque nouvel élément est inséré dans la partie suivante de manière circulaire. Par exemple `create_partitions( [1, 2, 3, 5, 8, 10, 14, 15, 18], 3)` renvoie `[[1, 5, 14], [2, 8, 15], [3, 10, 18]]`.

Solution :

```
def create_parts(E, k):
    n = len(E)
    assert n >= k
    P = []
    for i in range(k):
        P.append([])
    for i in range(n):
        P[i % 3].append(E[i])
    return P
```

A4. Écrire une fonction de prototype `barycentres(P)` qui renvoie la liste des barycentres de la partition représentée par `P`.

Solution :

1. c'est-à-dire en dessinant comme sur la figure 1

```
def barycentres(P):
    k = len(P)
    b = [0 for _ in range(k)]
    for i in range(k):
        assert len(P[i]) > 0
        for e in P[i]:
            b[i] += e
        b[i] /= len(P[i])
    return b
```

A5. Écrire une fonction de prototype `nearest_part(e, B)` dont les paramètres sont un élément `e` de l'ensemble `E` et la liste des barycentres `B` de la partition. Elle renvoie l'indice de la partie dont l'élément est le plus proche au sens de la distance euclidienne.

Solution :

Solution :

```
def nearest_part(e, B):
    index = 0
    dmin =  abs(e-B[0]) #dist(e, B[0]) en 1D !
    for j in range(1, len(B)):
        if abs(e - B[j]) < dmin:
            index, dmin = j, abs(e - B[j])
    return index
```

A6. Écrire une fonction de prototype `kmeans` qui implémente l'algorithme des K-moyennes en dimension 1. On s'appuiera sur les fonctions précédentes.

Solution :

```
def kmeans(data, k):
    n = len(data)
    assert n >= k
    P = create_partitions(data, k)
    while True:
        B = barycentres(P)
        new_P = [[] for _ in range(k)]
        for i in range(n):
            index = nearest_partition(data[i], B)
            new_P[index].append(data[i])
        # if P == new_P:
        # marche car à partir de la seconde itération même ordre  des
            échantillons dans les parties new_P
        #     moving = False
        # P = new_P
        # Version avec vérification des barycentres
        new_B = barycentres(new_P, E)
        if all(de(B[i], new_B[i]) < 1e-9 for i in range(k)):
```

```
                moving = False
            P = new_P
        return P
```

A7. Les enseignants aimeraient créer trois groupes de niveau identiques pour toutes les disciplines (mathématiques, physique, chimie, français, anglais, informatique, et sciences industrielles) en tenant compte des notes dans chaque discipline (et non pas de la moyenne comme précédemment). Proposer une solution à ce problème en décrivant les données d'entrées, les données de sorties ainsi que les structures de données utilisées.

Solution : Il suffit d'appliquer l'algorithme des K-moyennes avec k=3 mais avec les données suivantes :

Les données d'entrée sont un tableau Numpy de dimension `(n,7)`, c'est-à-dire autant de colonnes que de disciplines, autant de lignes que d'élèves.

Les données de sortie sont une partition de l'ensemble des élèves associant chaque élève à un groupe de TD. Cela pourrait être un tableau Numpy de dimension `(n,1)` ou une liste.

Il faudrait modifier pour cela le code précédent pour qu'il puisse travailler en dimension 7 (cf. TP!).

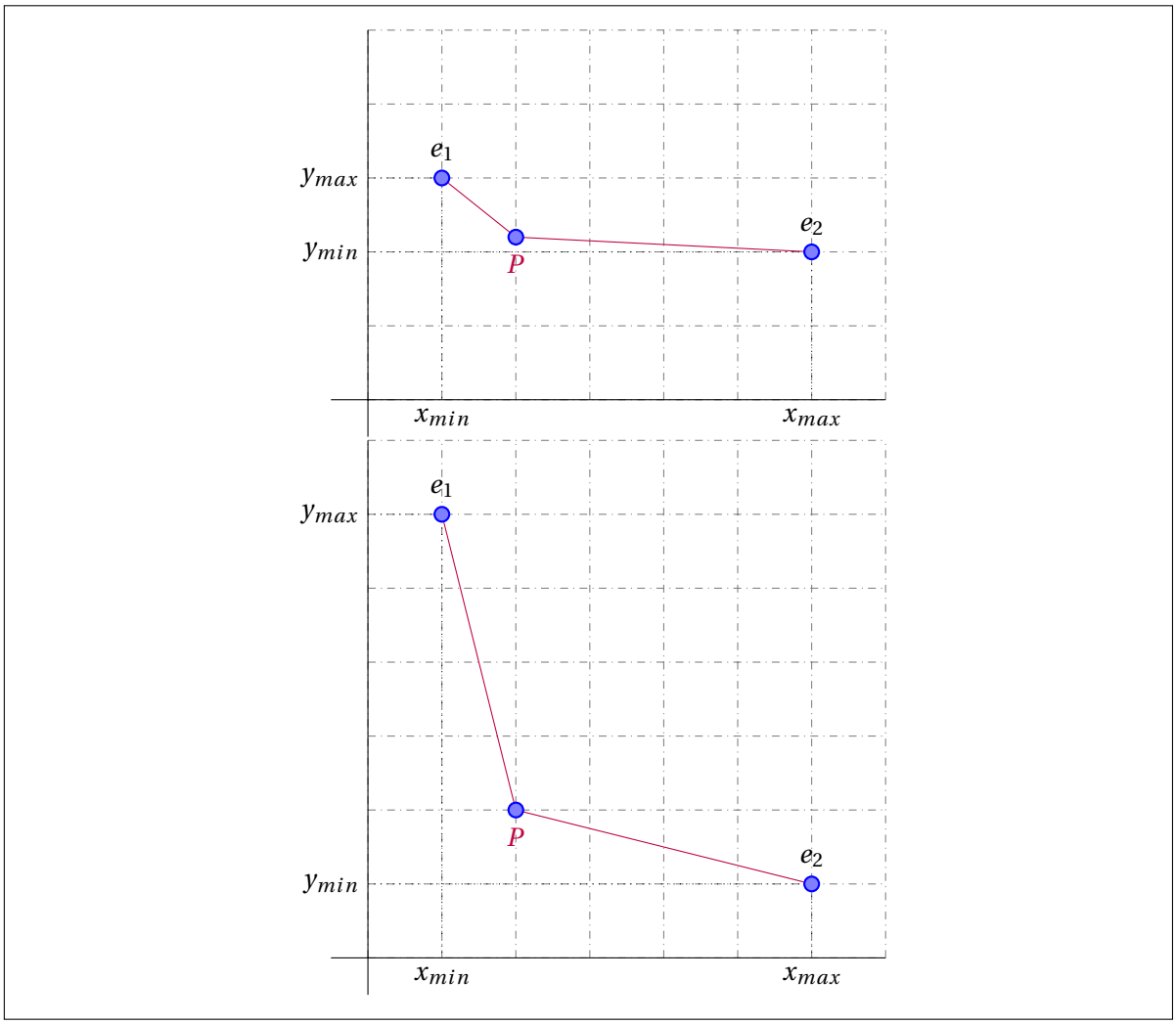
A8. Un collègue ayant déjà mis en place un tel algorithme dit : «il est nécessaire de normaliser les notes à l'entrée de l'algorithme, sinon il en résulte un biais dans la classification.» Ce collègue a-t-il raison? Pourquoi?

Solution : Si les notes ne sont normalisées, c'est-à-dire si les moyennes et écarts types ne sont pas tous les mêmes, alors les discplines pour lequelles l'écart type est faible vont avoir plus d'influence sur la valeur de la distance euclienne. Donc, il est nécessaire de ramener les données dans l'intervalles $[0,1]$ et de normaliser par rapport l'écart type.

Si les données ne sont pas mises à l'échelle, le paramètre dont la variation est la plus importante possède plus de poids proportionnellement qu'un autre dans la distance euclidienne et donc dans la classification opérée.

Ci-dessous, on a choisit un point P qui se situe proportionnellement à la même distance du minimum des deux paramètres x et y. Cependant, le premier graphique montre que la distance euclidienne accorde plus de poids à x qu'à y.

Le second graphique montre qu'après mise à l'échelle des données, le poids de chaque paramètre est identique et P est à égale distance de e_1 et e_2.



B Classification hiérarchique ascendante (Approche gloutonne)

On considère un ensemble d'entiers $E = \{x_0, \ldots, x_{n-1}\}$ **triés dans l'ordre naturel** ascendant des entiers. L'objectif est de partitionner cet ensemble en K classes mais en utilisant un autre algorithme que K-moyennes. Le critère de classification est toujours la proximité des moyennes des classes.

Une approche gloutonne pour résoudre ce problème procède comme suit :

- créer n classes distinctes, chacune contenant un seul élément de E,
- tant qu'il reste plus que K classes, fusionner les deux classes les plus proches, c'est-à-dire celles qui ont leurs moyennes les plus proches. En cas d'égalité, fusionner celles qui ont les moyennes les plus basses.

Dans cette sous-section, une partition est toujours une liste de listes.

B1. Écrire une fonction `moyenne(P,i)` qui calcule la moyenne de la classe d'indice i de P.

Solution :

Algorithme 1 Classification glouton

```
1 : Fonction P_GLOUTON(E, K)
2 :     n ← taille(E)
3 :     P ← la liste des singletons de E
4 :     tant que P comporte plus de K listes répéter
5 :         FUSIONNER_CLASSES(P, c)
6 :     renvoyer P
```

```
def moyenne(P, i):
    assert len(P) > 0
    assert len(P[i]) > 0
    m = 0
    for e in P[i]:
        m += e
    return m / len(P[i])
```

B2. Écrire une fonction `classe_lpp(P)` dont le paramètre est une partition `P`. La partition respecte l'ordre des entiers naturel également. Cette fonction renvoie l'indice `i_p` tel que C_{i_p} et $C_{i_{p+1}}$ sont les classes de `P` dont les moyennes sont les plus proches.

Solution : Le code ci-dessous s'appuie sur le fait que les entiers sont triés dans l'ordre naturel.

```
def classe_lpp(P):
    K = len(P)
    Lm = [0 for _ in range(K)]
    for i in range(K):
        Lm[i] = moyenne(P, i)
    i_p = 0
    mmin = Lm[i_p + 1] - Lm[i_p]
    for i in range(K - 1):
        if Lm[i + 1] - Lm[i] < mmin:
            mmin = Lm[i + 1] - Lm[i]
            i_p = i
    return i_p
```

B3. Écrire une fonction `fusion(P, i_p)` dont les paramètres sont une partition `P` de taille m et un indice `i_p`. Cette fonction renvoie une nouvelle partition de $m-1$ où les classes C_{i_p} et $C_{i_{p+1}}$ ont été fusionnées.

Solution :

```
def fusion(P, i_p):
    K = len(P)
    new_P = []
    for i in range(K - 1):
        if i < i_p:
            new_P.append(P[i])
```

```
            elif i == i_p:
                new_P.append(P[i] + P[i + 1])
            else:
                new_P.append(P[i + 1])
        return new_P
```

B4. En déduire une fonction `classes_glouton(E, K)` dont les paramètres d'entrée sont la liste `E` des éléments à partitionner et `K` le nombre de classes à créer. Cette fonction renvoie une partition de taille K sous la forme d'une liste de listes selon l'algorithme glouton.

Solution :

```
def classes_glouton(E, K):
    P = [[e] for e in E]
    while len(P) > K:
        i_c = classe_lpp(P)
        P = fusion(P, i_c)
    return P
```

B5. Quelle est la complexité de la fonction `classes_glouton` ?

Solution : Soit la taille de E. Comme la boucle `while` opère $n-K$ itérations, à chaque itération, la longueur de `P` décroît d'une unité. Les sous-fonctions ont pour complexité dans le pire des cas $O(n)$. La complexité de `classes_glouton` est en $O(n(n-K))$.

B6. Cette approche fournit-elle la partition optimale au sens des K-moyennes ?

Solution : Si on applique l'algorithme glouton à `E = [1, 2, 3, 5, 8, 10, 14, 15, 18]`, le résultat est : `P= [[1, 2, 3, 5, 8, 10, 14], [15], [18]]` ce qui est différent de la solution optimale trouvée précédemment. L'algorithme glouton n'est donc pas optimal. Par contre, il est plus rapide car de complexité moindre.

```
[[1], [2], [3], [5], [8], [10], [14], [15], [18]]
[[1, 2], [3], [5], [8], [10], [14], [15], [18]]
[[1, 2], [3], [5], [8], [10], [14, 15], [18]]
[[1, 2, 3], [5], [8], [10], [14, 15], [18]]
[[1, 2, 3], [5], [8, 10], [14, 15], [18]]
[[1, 2, 3, 5], [8, 10], [14, 15], [18]]
[[1, 2, 3, 5], [8, 10], [14, 15, 18]]
```