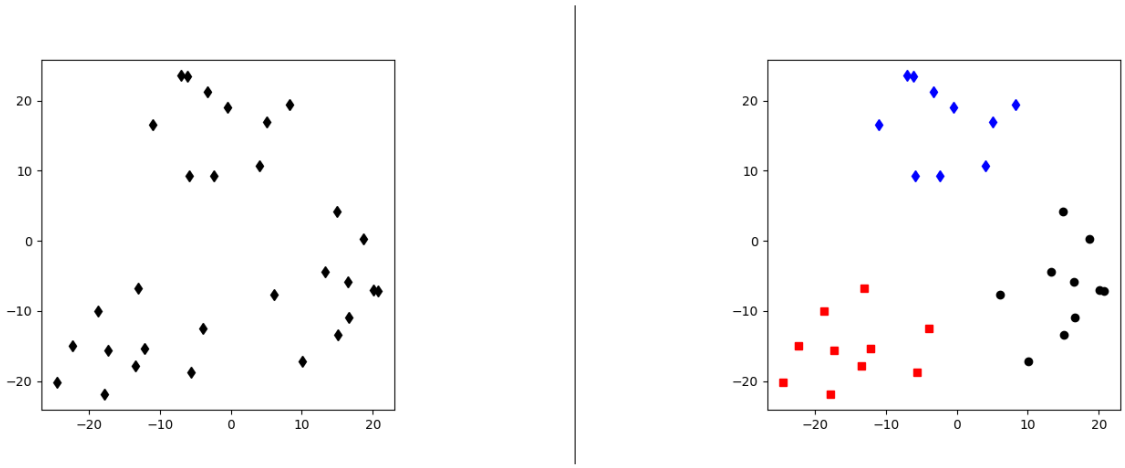
Apprentissage non supervisé

I Algorithme des k-moyennes

1. Description du problème

On considère des « points » que l'on souhaite répartir dans différentes catégories, en fonction de certaines de leurs caractéristiques.
Contrairement à l'algorithme des k plus proches voisins, on ne dispose pas d'un ensemble de points de référence ; on va seulement imposer le nombre de catégories que l'on souhaite créer.



Sur la figure de gauche, on voit qu'il semble raisonnable de regrouper les points en trois sous ensembles. On souhaiterait les regrouper comme sur la figure de droite.

2. Formalisation du problème

De façon plus formelle, on se donne un entier $k \in \mathbb{N}^*$ et on souhaite créer une partition de l'ensemble de points X qui minimise l'inertie du système.
Pour tout ensemble de points non vide $X_i = \{x_{i,j}, 1 \leqslant j \leqslant n_i\}$, on définit le barycentre de X_i (ou plutôt l'isobarycentre de X_i) par

$$B_i = \frac{1}{n_i} \sum_{j=1}^{n} x_{i,j}$$

On suppose que l'ensemble des points appartient à un espace euclidien dont la norme est notée $\| \|$.
Enfin, on note $\mathcal{P}_k$ l'ensemble des partitions de l'ensemble X en k sous-ensembles non vides.
L'inertie d'une partition $P = \{X_1, \dots, X_k\} \in \mathcal{P}_k$, donc la quantité que l'on souhaite minimiser, est définie par

$$I_P = \sum_{i=1}^{k} \sum_{x \in X_i} \|B_i - x\|$$

3. L'algorithme des k-moyennes

Pour tenter de minimiser cette quantité, on va procéder de la façon suivante :

- on crée une première partition aléatoire de X en k sous ensembles non vides
- on détermine les barycentres B_i, $1 \leqslant i \leqslant k$, de chacun des sous ensembles de cette partition
- on crée une nouvelle partition de X en k sous ensembles en regroupant les points de X en fonction du point B_i dont ils sont le plus proche.
- on recommence au deuxième point avec cette nouvelle partition jusqu'à ce que le système n'évolue plus, c'est-à-dire jusqu'à ce que la nouvelle partition soit égale à la précédente.

On peut démontrer que l'inertie du système est décroissante au cours du déroulement de cet algorithme, ce qui assure qu'il se terminera bien puisque l'ensemble des partitions de X est fini.
Si la convergence est trop lente, on peut choisir d'interrompre la recherche après un certain nombre d'itérations ou bien lorsque l'inertie du système ne varie pas beaucoup (en dessous d'un certain seuil) au cours d'une ou plusieurs itérations consécutives.

L'avantage principal de cet algorithme est sa simplicité de mise en œuvre.
Mais il possède plusieurs défauts :

— Cet algorithme ne fournit pas toujours une solution optimale, donc une partition qui minimise l'inertie du système, mais converge vers une configuration qui réalise un minimum local de l'inertie.

— La solution donnée par cet algorithme dépend de la première partition créée. Si on utilise deux fois de suite cet algorithme avec le même ensemble X, on peut avoir deux partitionnements différents à la fin selon le choix qui a été fait au moment de créer la première partition. une des deux solutions peut correspondre à un minimum local et la deuxième à un autre minimum local (ou au minimum absolu que l'on recherche).

— Pour fonctionner, on doit connaître à l'avance le nombre de classes à créer : dans le cas où ce nombre est inconnu, on peut être amené à faire des tests avec différentes valeurs de k pour déterminer celle qui est la plus satisfaisante.

II Codage de l'algorithme sur un exemple

On suppose disposer d'un ensemble X de points du plan.
On a pour, commencer, besoin d'une distance euclidienne : dans le plan, on utilisera la norme euclidienne canonique sur $\mathbb{R}^2$.

```
def d(X,Y) :
    return ((X[0]-Y[0])**2+(X[1]-Y[1])**2)**0.5
```

Il nous faut ensuite une fonction pour calculer le barycentre d'une partie `Y` non vide de points du plan. On peut noter que les barycentres ne sont en général pas des points de l'ensemble X lui même.

```
def barycentre(Y) :
    s1,s2 = 0,0
    for (x,y) in Y :
        s1 += x
        s2 += y
    return s1/len(Y),s2/len(Y)
```

Puis une fonction qui, pour un point `p` du plan et une partie `M` de points (qui seront les barycentres dans l'utilisation de cette fonction), détermine l'indice d'un point de `M` dont `p` est le plus proche

```
def plusProche(p,M) :
    ind,dist = 0,d(p,M[0])
    for i in range(len(M)) :
        point = M[i]
        dist1 = d(p,point)
        if dist1 < dist :
            ind = i
            dist = dist1
    return ind
```

Enfin, une fonction permettant de créer aléatoirement la première partition de X. La fonction `shuffle` du module `random` permet de mélanger une liste, la fonction `randint(a,b)` du module `random` renvoie un entier aléatoire de l'ensemble $[\![a, b]\!]$ (bornes incluses). Il faut faire attention à créer une partition de X en k sous ensembles non vides

```
import random as rd

def initialisation(X,k) :
    rd.shuffle(X)
    part = [[] for _ in range(k)]
    # on place un point dans chaque sous ensemble
    for i in range(k) :
        part[i].append(X[i])
    # on répartit aléatoirement les autres
    for i in range(k,len(X)) :
        part[rd.randint(0,k-1)].append(X[i])
    return part
```

On peut alors coder l'algorithme

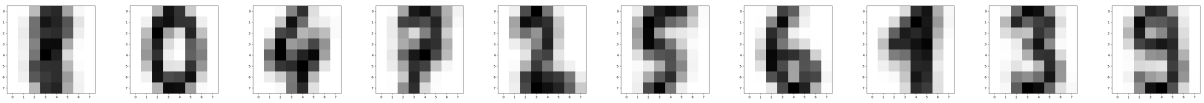
```
def kMoy(X,k) :
    n = len(X)
    part = initialisation(X,k)
    evolue = True
    while evolue :
        B = [barycentre(E) for E in part]
        part2 = [[] for _ in range(k)]
        for i in range(n) :
            ind = plusProche(X[i],B)
            part2[ind].append(X[i])
        if part == part2 :
            evolue = False
        part = part2
    return part
```

Cet algorithme peut ne pas fonctionner correctement car il peut, en dehors de l'initialisation, créer une partition dont un sous ensemble est vide, ce qui provoque une erreur au moment du calcul de barycentre mais ceci ne se produit pas général, si les points peuvent effectivement se répartir en k classes. Pour corriger ce problème, on peut par exemple placer dans le sous ensemble vide un point dont la contribution dans la valeur de l'inertie est la plus grande (un point x pour lequel $\|x - B_i\|$ est maximal).

III Exemples d'utilisation

1. Reconnaissance de chiffres

Si on considère un ensemble d'image représentant les 10 chiffres, on peut appliquer cet algorithme (donc avec $k = 10$) pour regrouper les images. Si à l'issue de cet algorithme, on affiche les images (de 8×8 pixels) correspondant aux barycentres des sous ensembles créés, on peut obtenir les dessins suivants :



Si on cherche la proportion d'images d'une classe qui correspondent effectivement au bon chiffre, on obtient les résultats suivants :

chiffre	0	1	2	3	4	5	6	7	8	9
résultat	99%	57%	84%	85%	98%	87%	97%	86%	46%	58%

Le plus mauvais résultat est obtenu pour le chiffre 8, qui donne effectivement l'image la plus « floue » .
Ces résultats sont susceptibles de varier si on exécute à nouveau l'algorithme puisque l'initialisation de la partition sera sans doute différente.

2. Compression d'image

Une image couleur de h lignes et ℓ colonnes est codée en Python par une matrice (tableau `Numpy`) de h lignes et ℓ colonnes dont chaque élément donne la couleur d'un pixel ; la couleur d'un pixel étant codée par un triplet `R,G,B`, donnant la « quantité » de rouge, vert et bleu de la teinte. Chacune de ces couleurs « primaire » `R,G,B` est codée sur un octet (8 bits), ce qui donne $2^8 = 256$ possibilités pour chaque couleur primaire donc $256^6 = 16777216$ couleurs disponibles. Une image nécessite donc $h \times \ell \times 24$ bits en mémoire.
Si on limite le nombre de couleurs utilisées à 16 par exemple, il suffira de 4 bits ($2^4 = 16$) pour repérer la couleur de chaque pixel donc un espace de $h \times \ell \times 4$ bits pour stocker l'image (mais qui sera de moins bonne qualité), en négligeant l'espace nécessaire à la mémorisation des 16 couleurs utilisées (16×24 bits).

Pour déterminer ces 16 couleurs à utiliser, on peut appliquer l'algorithme des k moyennes (avec $k = 16$) de façon à déterminer les couleurs les plus « utiles » qui seront les barycentres des 16 parties créées.
Dans l'exemple suivant l'image initiale (de 640×427 pixels) comporte 75792 couleurs alors que celle de droite n'en nécessite plus que 16.

Image initiale

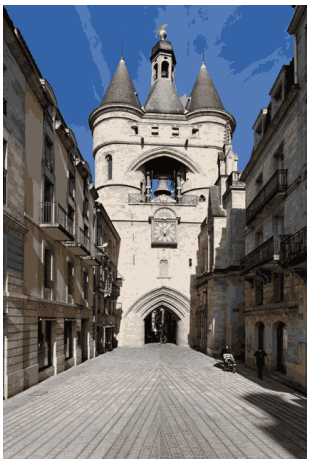


Image avec 16 couleurs