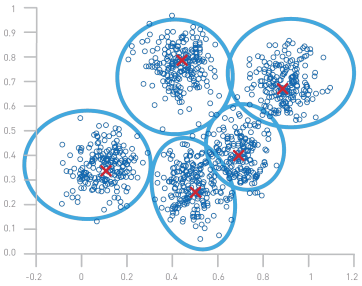
Chapitre 7

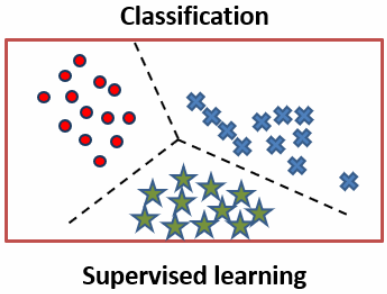
ALGORITHME DES k-MOYENNES

Toujours dans le thème « Intelligence Artificielle », dans ce chapitre on va essayer d'apprendre à l'ordinateur à trier un ensemble de données brutes en les répartissant en plusieurs paquets (k paquets précisément), appelés clusters. Cette fois-ci on ne disposera pas de jeu de données de référence par rapport auxquelles on pourra chercher les similitudes.

Il va donc s'agir de trouver comment les données peuvent se regrouper ou pas autour d'une même mesure de référence qui les caractérise. Sur le graphique ci-dessous on voit bien se détacher assez nettement 5 paquets de données, on peut notamment placer le point moyen (barycentre) de chacun des 5 nuages de points.

Le problème c'est comment apprendre à l'ordinateur à opérer ce regroupement ? L'algorithme des k-moyennes répond de façon assez simple à ce problème, mais pas nécessairement de façon optimale et unique comme nous pourrons le constater.





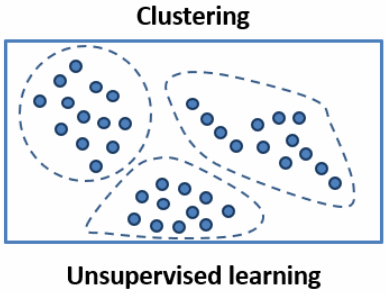


Table des matières

Le but des algorithmes présentés dans cette partie du cours est **classer automatiquement nos données** en différents groupes.

Dans l'algorithme des k plus proches voisins vu au cours précédent, on connait d'avance les différentes sortes de données (ex : chat, chien, cheval) et on possède un ensemble d'apprentissage, qui est un ensemble de données déjà étiquetées. On utilise cet ensemble d'apprentissage pour déterminer la classe/étiquette d'une nouvelle donnée (on prend l'étiquette majoritaire parmi les k plus proches voisins).

Problème : On peut vouloir classer automatiquement nos données en différents groupes sans savoir d'avance quels types de groupes (Ex : une plateforme de films veut regrouper ses utilisateurs selon la similiarité de leur profil). Ou alors on peut savoir quels types de groupes on voudrait avoir, mais on ne dispose pas de données déjà étiquetées (Ex : reconnaissance automatique de chiffres comme fait au cours prédent, mais sans avoir des images dont on sait déjà à quel chiffres elles correspondent).

Dans ces deux types de cas, on ne peut pas faire d'apprentissage supervisé, mais on peut faire de l'**apprentissage non supervisé** (c'est-à-dire qui n'utilise pas de données déjà étiquetées).

Nous allons demander à l'ordinateur de séparer les données en k groupes (appelés **clusters**) par proximité. Les éléments des k groupes formés devront avoir le maximum de ressemblances entre eux et de différences avec les éléments des autres groupes.

L'entier k, qui correspond au nombre de groupes, sera imposé par l'utilisateur. Si on prend l'exemple de la reconnaissance de chiffres, on va choisir ici $k = 10$ dans la mesure où les images sont censées représenter les 10 chiffres. Dans d'autres cas d'application, on n'aura pas d'a priori sur la bonne valeur de k, il faudra en tester plusieurs.

I BARYCENTRE D'UN CLUSTER

Plus généralement, partons du principe que l'on ait ces k **clusters**, alors on peut calculer le **barycentre** de chacun de ces clusters.

Réciproquement, si on avait les barycentres de chacun de ces groupes, on pourrait reconstituer les clusters, chaque élément allant dans le cluster dont le barycentre est le plus proche de cet élément parmi tous les éléments.

- on note C_1, C_2,..., C_k ces clusters,
- on note $b_i = \frac{1}{|C_i|} \sum_{c \in C_i} c$ le barycentre du i-ième cluster C_i (et $m_i = \sum_{c \in C_i} \|c - b_i\|^2$ son moment d'inertie).

Dans le cas de notre exemple de reconnaissance de chiffres, où les données sont des images de 8×8 pixels, on peut écrire ainsi la fonction qui calcule le barycentre d'un cluster (en supposant que la variable `cluster` contient les indices de position des images du cluster dans le jeu de données `Images`). Le barycentre est lui-même une image de 8×8 pixels.

```
def Barycentre(cluster):
    B=[[0 for j in range(8)] for i in range(8)] # Initialisation du barycentre à 0
    for i in range(8):
        for j in range(8): # pour chaque pixel de l'image
            for k in cluster: # pour chaque image du cluster
                B[i][j]=B[i][j]+Images[k][i][j] # on somme l'intensité du pixel
            B[i][j]=B[i][j]/len(cluster) # on renormalise pour obtenir une moyenne
    return B
```

Principe : pour chaque pixel de l'image, on calcule la valeur moyenne de l'intensité de ce pixel parmi les images du cluster.

L'idéal serait d'obtenir des clusters avec $\sum_{i=1}^{k} m_i$ minimale, donc chercher $\min_{\text{partitions de } C} \left(\sum_{i=1}^{k} \left(\sum_{c \in C_i} \|c - \frac{1}{|C_i|} \sum_{c \in C_i} c\|^2 \right) \right)$

Néanmoins ceci est un problème de minimisation trop difficile à programme en un temps raisonnable !

II PLUS PROCHE BARYCENTRE

Nous allons procéder de la façon suivante. Pour initialiser nous allons :

- choisir une répartition aléatoire en k clusters (et pour chaque cluster nous calculons son barycentre.)
- OU choisir k barycentres de façon aléatoire parmi nos données (et former des clusters de départ avec le principe ci-après).

Puis nous allons répéter ces deux étapes jusqu'à convergence (= stabilisation des clusters) :

- Pour chaque donnée, nous allons la rattacher à un cluster en cherchant le barycentre le plus proche.
- Pour chaque cluster nous recalculons son barycentre.

La première étape peut-être réalisée grâce à la fonction suivante, la seconde avec la fonction `Barycentre`.

```
def PlusProcheBarycentre(B,I):
    dmin=np.inf
    for j in range(len(B)):
        if Dist(I,B[j])<dmin:
            jmin,dmin=j,Dist(I,B[j])
    return jmin
```

```
def Dist(Img1,Img2):
    S=0
    for i in range(8):
        for j in range(8):
            S=S+(Img1[i][j]-Img2[i][j])**2
    return S**(1/2)
```

Remarque. Le choix initial des barycentres/clusters est aléatoire, et influe sur le résultat.

III L'ALGORITHME DES k-MOYENNES

Faisons le choix de stocker les barycentres initialement et à chaque étape. A l'aide d'une boucle `while` nous allons réaliser ces deux étapes. Nous allons comparer l'ancienne liste des barycentres à la nouvelle et arrêter l'algorithme dès que celle-ci n'est plus modifiée.

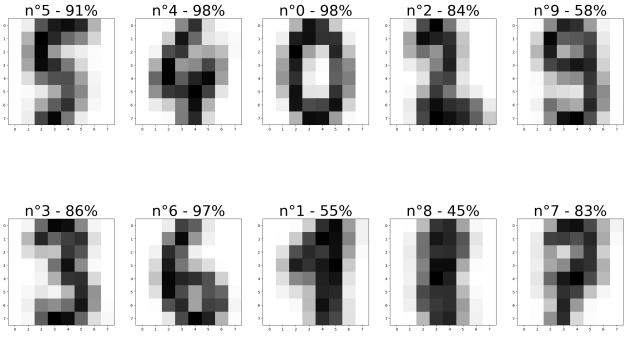
```
def kMoyennes(data,k):
    B=[data[np.random.randint(len(data))] for i in range(k)] # liste des barycentres
    while True:
        Clusters=[[] for i in range(k)] # Initialisation des k clusters vides
        for i in range(len(data)):  # pour chaque donnée
            jmin=PlusProcheBarycentre(B,data[i]) # on calcule le plus proche barycentre
            Clusters[jmin].append(i) # on ajoute la donnée dans le bon cluster
        newB=[Barycentre(Clusters[j]) for j in range(k)] # calcul nouveaux barycentres
        if newB==B: # Si les barycentres ne bougent plus
            return Clusters # arrêt de la fonction, renvoi des clusters obtenus
        B=newB # sinon on remplace les anciens barycentres par les nouveaux
```

On admet que **cet algorithme termine bien** (preuve hors-programme). C'est-à-dire que quelles que soient les données de départ, on arrivera à une étape où les nouveaux barycentres sont identiques aux anciens.

Cet algorithme ne converge pas vers un minimum global mais seulement vers un minimum local.

D'ailleurs, on obtient des résultats différents si on lance plusieurs fois cet algorithme. Ceci traduit la dépendance dans l'aléatoire des barycentres de départ.

Ici nous avons un algorithme dit **non supervisé** : il n'a pas utilisé de données étiquetées. Notre programme a tout de même réussi à regrouper les données en 10 paquets qui correspondent plus ou moins à des groupes de même chiffre.



Les 10 barycentres des 10 clusters et leur étiquette majoritaire, ainsi que le % de cette étiquette majoritaire au sein du cluster.

IV EXERCICES

On rappelle qu'on peut traiter les images en Python en les voyant comme une liste de listes (ou matrices). Ainsi, si on note `Img` cette image, `Img[i][j]` représente la couleur du pixel situé à la ligne `i` et la colonne `j`.

Pour une image en couleur, `Img[i][j]` est alors une couleur, représentée par une liste de trois nombres : un niveau de rouge (R), un niveau de vert (G) et un niveau de bleu (B), ces trois nombres sont des entiers entre 0 et 255 et ceci forme le code RGB de la couleur.

Etant donnée une image en couleurs, on cherche à regrouper les pixels en k clusters, de façon à ce que dans chaque cluster, les pixels aient des couleurs semblables. Pour cela on mettra en place l'algorithme des k-moyennes. Ensuite on remplacera les valeurs des pixels de chaque cluster par la valeur du barycentre de chaque cluster, dans le but de réduire le nombre de couleurs (compression de l'image).

Les clusters contiendront les coordonnées `(i,j)` des pixels.

1. Télécharger l'image `Hooke_Bernoulli.jpg` et le fichier `TP_07.py` qui charge et affiche l'image avec les commandes suivantes, et exécuter sans modifier

```
import matplotlib.pyplot as plt
import imageio
Img=imageio.imread("Hooke_Bernoulli.jpg").tolist()
# Convertit l'image Hooke_Bernoulli.jpg en liste
plt.figure() # Création d'une figure
plt.imshow(Img) # Commande pour dire qu'on veut afficher Img
plt.title("Qu'ils␣sont␣mignons...") # Titre
plt.show() # Affichage final

n=len(Img)
p=len(Img[0])
```

2. Combien de couleurs différentes y a-t-il dans l'image ? (parcourir tous les pixels)
3. Créer une fonction `Barycentre(cluster)` qui va renvoyer la couleur moyenne d'un cluster de couleurs. Si jamais le cluster est vide, on renverra une couleur au hasard, donc un triplet de 3 entiers entre 0 et 255. (Rappel : la commande `np.random.randint(n)` renvoie un nombre de façon uniforme entre 0 et $n-1$).
4. Créer une fonction `PlusProcheBarycentre(B,color)`, où `B` est une liste de barycentres et `color` une couleur, qui va renvoyer l'indice du barycentre le plus proche de `color` dans la liste `B` (s'inspirer du cours).
5. Écrire la fonction `kMoyennes(data,k)` qui effectue l'algorithme des k-moyennes comme décrit au début de l'exercice (s'inspirer du cours).
6. En utilisant les clusters fournis par la fonction `kMoyennes` (avec $k = 10$ couleurs par exemple), modifier l'image, pour que la couleur de chaque pixel soit modifiée pour valoir celle du barycentre du cluster auquel appartient le pixel.

Remarque.
Cela permet de stocker qu'une seule fois chaque couleur et après, il n'y a plus qu'à mémoriser l'ensemble des pixels qui ont cette couleur et donc de minimiser la mémoire prise par le stockage (compression).

Remarque.
On remarque que pour une centaine de couleurs, l'œil humain ne fait quasiment plus de différence avec l'image d'origine qui contenait près de 8 000 couleurs. Il n'y a donc plus qu'à conclure qu'acheter une nouvelle télévision avec des millions de couleurs est absurde et non écologique...

En fait les scientifiques estiment qu'en moyenne, un être humain peut faire la distinction entre plus d'un million de couleurs différentes. Mais, cette faculté varie d'une personne à l'autre. Certaines personnes ne voient que quelques centaines de couleurs différentes, alors que d'autres peuvent en voir jusqu'à 100 millions ! En moyenne, des études plutôt qu'un nombre maximal de couleurs discernables autour de 300 000 *est plus réaliste.*