TP8 - Algorithmes d'apprentissage non supervisé

-

Les k moyennes

I. Introduction

Dans ce TP, on souhaite mettre en place l'algorithme de k moyennes qui est un algorithme d'*apprentissage non supervisé*. On rappelle qu'il consiste à faire en sorte qu'à partir d'un ensemble de données, la machine soit capable de créer ses propres catégories, et si possible, que ces catégories soient pertinentes pour nous. Ce modèle est plus proche de notre propre modèle d'apprentissage, basé sur l'observation.

On s'intéresse donc maintenant au problème suivant : comment comprendre la structure d'un jeu de données sans aider la machine en lui donnant un jeu d'apprentissage étiqueté ?
En pratique, il s'agit pour la machine de regrouper les données proches au sein d'une même classe, à charge pour l'humain d'interpréter ensuite ce regroupement.

Par exemple, en regardant la Figure 1, les données semblent se regrouper en 4 classes. On souhaite donc que la machine en détecte 4 elle aussi. Pour cela, on impose à la machine le nombre k de classes. On appelle $C_1, \dots, C_k$ ces classes et on note, pour tout $j \in [\![1, k]\!]$:

- μ_j le **barycentre** de C_j. Autrement dit : $\boxed{\mu_j = \dfrac{1}{\mathrm{Card}(C_j)} \displaystyle\sum_{x \in C_j} x}$;
- m_j le **moment d'inertie** de C_j. Autrement dit : $\boxed{m_j = \displaystyle\sum_{x \in C_j} \|x - \mu_j\|^2}$.

Pour définir au mieux les k classes, on cherche donc à minimiser la somme des moments d'inertie :

$$\boxed{I = \sum_{j=1}^{k} m_j}$$

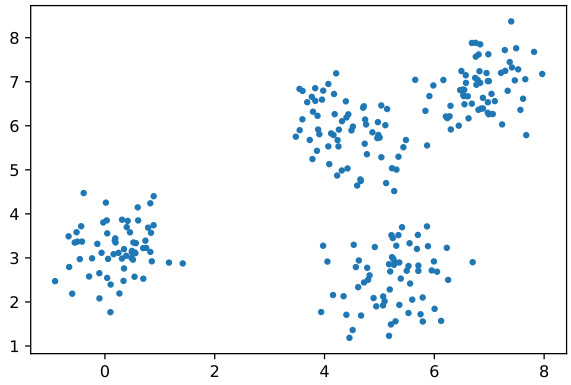

Fig. 1 Données

II. L'algorithme des k moyennes

Le calcul de la classification optimale, c'est-à-dire en minimisant la somme des moments d'inertie est un problème très coûteux en temps. On penche donc pour l'utilisation d'un algorithme glouton. Ce dernier assurera d'obtenir une « bonne » solution, mais pas forcément la meilleure.

L'**algorithme des k moyennes** consiste à :

1) choisir aléatoirement k points μ_1, ..., μ_k, qui contiendront, à terme, les centres des clusters (ou classes) construits ;

2) associer chacun des points du nuage au centre μ_j le plus proche. On crée ainsi k clusters C_1, ..., C_k ;

3) calculer les barycentres μ_1, ..., μ_k de chacun des clusters C_1, ..., C_k qui remplacent les valeurs précédentes ;

4) reprendre le calcul à partir de l'étape *2)*.

On admet, dans le programme de TPC, que cet algorithme converge. Autrement dit, on admet qu'à partir d'une certaine étape, les centres μ_1, ..., μ_k ne sont plus modifiés et donc que les classes C_1, ..., C_k sont stabilisées.
On rappelle, qu'en temps qu'algorithme glouton, l'algorithme des k moyennes renvoie une configuration pour laquelle la somme des moments d'inertie est un minimum local, mais pas forcément le minimum global.

III. Un exemple d'apprentissage non supervisé

III.1. Préambule

► Récuperer le fichier `donnees_k-moyennes.py` sur ma page.

Ce fichier contient deux ensembles de points, `data1` et `data2`, qui serviront d'exemples dans ce TP. Ces ensembles de points sont stockés sous forme de listes de listes à deux éléments (abscisse et ordonnée du point).
Que contient donc `data1[i]` ? `data1[i][0]` ? `data1[i][1]` ?

> Comme décrit dans l'énoncé, la commande `data1[i]` contient une liste à 2 éléments :
> × d'une part `data1[i][0]` contient l'abscisse du $(i+1)^{\text{ème}}$ point du jeu de données,
> × d'une part `data1[i][1]` contient l'ordonnée du $(i+1)^{\text{ème}}$ point du jeu de données,

► Le fichier `donnees_k-moyennes.py` contient aussi la définition d'une fonction `affiche` qui prend indifféremment pour argument une liste de points ou une liste de listes de points formant une partition de l'ensemble du jeu de données.

- Si le paramètre d'entrée est une liste de points, alors la fonction affiche à l'écran tous les points en une même couleur.
- Si le paramètre d'entrée est une liste de listes de points, alors chaque sous-liste formant la partition sera affichée avec une couleur différente.

Quelle commande permet d'afficher le nuage de points des données de `data1` ?

> Il s'agit de la commande : `affiche(data1)`

III.2. L'algorithme des k moyennes

- Écrire une fonction `barycentre` qui prend pour argument un ensemble de points sous forme de liste et renvoie les coordonnées du barycentre de ces points.

```
def barycentre(data) :
    ''barycentre(data : list) -> list''
    x = 0
    y = 0
    for d in data :
        x += d[0]
        y += d[1]
    return [x / len(data), y / len(data)]
```

- Écrire une fonction `dist` qui prend en argument deux point `p` et `q` sous forme de listes et renvoie le carré de la distance euclidienne de `p` à `q`.

```
def dist(p, q) :
    ''dist(p : list, q : list) -> float''
    return (p[0] - q[0])**2 + (p[1] - q[1])**2
```

- Écrire une fonction `plusProche` qui prend en argument un point `p` sous forme de liste et un ensemble de points $\mu = (\mu_1, \dots, \mu_k)$ stockée dans une variable `mu` sous forme de liste de listes, et renvoie l'indice `i` $\in [\![0, k-1]\!]$ correspondant au point μ_i le plus proche de `p`.

```
def plusProche(p, mu) :
    ''plusProche(p : list, mu : list(list)) -> imin : int''
    dmin = dist(p, mu[0])
    imin = 0
    k = len(mu)
    for i in range(k) :
        d = dist(p, mu[i])
        if d < dmin :
            dmin = d
            imin = i
    return imin
```

- Exécuter le script suivant.

```
1  L = [x**2 for x in range(1000)]
2  mu = sample(L, 7)
```

Que contiennent les variables `L` et `mu` ?

- La variable `L` contient tous les carrés des entiers de 0 à 999.
- La variable `mu` contient 7 éléments distincts de `L` choisis aléatoirement (uniformément).

En déduire ce que revoie la commande `sample(data, k)` où `data` est un jeu de données sous forme de liste de listes, et `k` est un entier.

La commance `sample(mu, k)` renvoie `k` éléments distincts choisis aléatoirement parmi les éléments de `mu`.

- Écrire une fonction `kmeans` qui prend en paramètre un ensemble de points `data` sous forme de liste de listes, et un entier `k`, et qui renvoie la partition en `k` clusters du jeu de données à l'aide de l'algorithme des k moyennes.
Cette partition sera stockée sous la forme d'une liste de `k` listes de points (on rappelle que les points sont stockés sous la forme d'une liste à 2 éléments). Cela permettra de visualiser la classificaiton des points du jeu de données à l'aide de la fonction `affiche`.
Pour éviter d'obtenir des clusters vides lors de la $1^{ère}$ *étape de l'algorithme des k moyennes (ce qui provoquerait une erreur lors d'une calcul du barycentre), on prendra pour valeurs initiales de* μ_1, ..., μ_k *lors de l'étape **1)**,* `k` *points distincts parmi les données à traiter (ce qui garantit que chaque clister contient au minimum un point). On pourra utiliser la fonction* `sample` *étudiée précédemment.*

On rappelle que l'algorithme se termine lorsque les barycentres ne sont plus modifiés lors de l'étape ***3)***. On propose alors le script suivant.

```
1  def kmeans(data, k) :
2      '' kmeans(data : list(list), k : int) -> clusters : list(list)''
3      mu = sample(data, k)
4      while True :
5          clusters = [[] for _ in range(k)]
6          for p in data :
7              i = plusProche(p, mu)
8              clusters[i].append(p)
9          newmu = [barycentre(clusters[i]) for i in range(k)]
10         if mu == newmu :
11             return clusters
12         mu = newmu
```

On obtient par exemple les classifications suivantes, pour $k = 4$. On remarque que les partitions

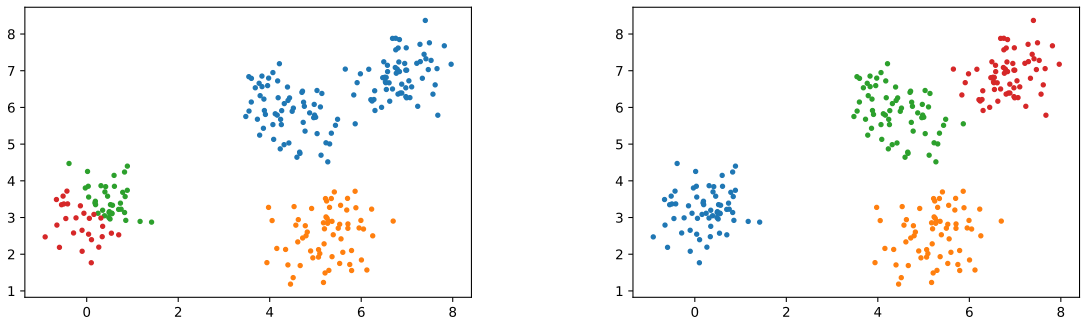

Fig. 2 Classification par algorithme des k moyennes

obtenus ne sont pas forcément optimaux. On va donc chercher à améliorer l'algorithme des k moyennes.

III.3. Une amélioration de l'algorithme

L'un des objectifs de l'algorithme des k moyennes est de minimiser l'**inertie** de la partition $(C_1, \dots, C_k)$ obtenue. On souhaite donc minimiser la quantité :

$$I = \sum_{j=1}^{k} \left(\sum_{p \in C_j} \mathrm{d}(p, \mu_j) \right)$$

où μ_j désigne toujours le barycentre du cluster C_j.

- Écrire une fonction `inertie` qui prend en argument une liste de clusters `clusters` et qui renvoie l'inertie de cette partition.

```
def inertie(clusters) :
    '' inertie(clusters : list(list)) -> I : float''
    I = 0
    for data in clusters :
        mu = barycentre(data)
        for p in data :
            I += dist(p, mu)
    return I
```

- On sait que l'algorithme des k moyennes fournit une partition qui correspond à un minimum local de l'inertie, mais pas forcément au minimum global. Pour augmenter les chances d'obtenir ce minimum global, on peut appliquer 20 fois de suite l'algorithme des k moyennes pour ne garder que la réponse d'intertie minimale.
  Écrire alors une fonction `bestKmeans` qui prend en argument un jeu de donnée `data` sous forme de liste de listes et un entier `k`, et qui renvoie la meilleure de ces 20 partitions (celle qui a la plus petite inertie).

```
def bestKmeans(data, k) :
    ''bestKmeans(data : list(list), k : int) -> clustersmin = list(list)''
    clustersmin = kmeans(data, k)
    mini = inertie(clustersmin)
    for _ in range(1, 20) :
        clusters = kmeans(data, k)
        inert = inertie(clusters)
        if inert < mini :
            mini = inert
            clustersmin = clusters
    return clustersmin
```

IV. Le choix de l'entier k

- Quelles commandes permettent d'afficher la partition (on parle aussi de **clustering**) de l'ensemble de données `data2` pour $k = 3$ et $k = 4$?

```
cluster3 = kmeans(data2, 3)
cluster4 = kmeans(data2, 4)
affiche(cluster3)
affiche(cluster4)
```

On aimerait savoir quelle est la meilleure partition parmi les 2 précédentes. Pour cela, on introduit la notion de **silhouette** d'un point.

- Auparavant, écrire une fonction `moyenne` qui prend en paramètre une liste `L` et renvoie la moyenne des nombres contenus dans cette liste.

```
def moyenne(L) :
    ''moyenne(L : list) -> float''
    return sum(L) / len(L)
```

▶ Pour définir la **silhouette** d'un point p appartenant au cluster C_i d'une partition $(C_1, \dots, C_k)$, on introduit successivement les quantités suivantes :

× le réel $a(p)$ est la moyenne des distances entre p et les points q appartenant au même cluster que p :

$$\boxed{a(p) = \frac{1}{\mathrm{Card}(C_i) - 1} \sum_{q \in C_i \setminus \{p\}} \mathrm{d}(p, q)}$$

× le réel $b(p)$ est la moyenne des distances entre p et les points q du cluster le plus proche :

$$\boxed{b(p) = \min_{j \neq i} \left(\frac{1}{\mathrm{Card}(C_j)} \sum_{q \in C_j} \mathrm{d}(p, q) \right)}$$

× la silhouette de p est alors la quantité :

$$\boxed{\mathrm{sil}(p) = \frac{b(p) - a(p)}{\max\big(a(p), b(p)\big)}}$$

Par construction, la silhouette de p est un réel compris entre 1 et -1. De plus :

× si $\mathrm{sil}(p)$ est proche de 1, alors $a(p)$ est négligeable devant $b(p)$. Or :

- une petite valeur de $a(p)$ illustre le fait que p est bien « intégré » dans son cluster,
- une grande valeur de $b(p)$ illustre le fait que p n'est pas bien intégré dans les autres clusters.

En résumé, une valeur de la silhouette proche de 1 démontre que le cluster associé à p est bien choisi.

× si $\mathrm{sil}(p)$ est proche de -1, alors $b(p)$ est négligeable devant $a(p)$, et donc le cluster attribué à p est mal choisi.

Écrire maintenant successivement les fonctions `a`, `b` et `sil` qui prennent en paramètre un point `p` sous forme de liste, et une partition `clusters` sous forme de liste de listes, et qui renvoie la silhouette de `p` dans la partition `clusters`.

```
def a(p, clusters) :
    ''a(p : list, clusters : list(list)) -> float''
    j = 0
    while p not in clusters[j] :
        j += 1
    return moyenne( [dist(p, q) for q in clusters[j] if q != p] )

def b(p, clusters) :
    ''b(p : list, clusters : list(list)) -> smin : float''
    smin = float('inf')
    for clust in clusters :
        if p not in clust :
            smin = min(smin, moyenne( [dist(p, q) for q in clust] ))
    return smin
```

```python
def sil(p, clusters) :
    ''p : list, clusters : list(list)) -> float''
    ap = a(p, clusters)
    bp = b(p, clusters)
    return (bp - ap) / max(ap, bp)
```

- La silhouette d'une partition $(C_1, \ldots, C_k)$ est la moyenne des moyennes des silhouettes des points de chaque cluster. Autrement dit :

$$\text{sil}(C) = \frac{1}{k} \sum_{i=1}^{k} \left(\frac{1}{\text{Card}(C_i)} \sum_{p \in C_i} \text{sil}(p) \right)$$

Écrire une fonction `silhouette` qui prend en paramètre une partition `clusters` et qui calcule la silhouette d'une partition.

```python
def silhouette(clusters) :
    return moyenne( [moyenne( [sil(p, clusters) for p in c] ) for c in clusters] )
```

- Quelle commande permet de comparer la partition du jeu de données `data2` obtenu avec l'algorithme des k moyennes pour $k = 3$, et pour $k = 4$?
Quelle est la meilleure partition ?

```python
bestcluster3 = silhouette(bestKmeans(data2, 3))
bestcluster4 = silhouette(bestKmeans(data2, 4))
print([bestcluster3, bestcluster4])
```

On obtient :

× une silhouette d'environ 0.81 pour $k = 3$,

× une silhouette d'environ 0.71.

La partition pour $k = 3$ est donc meilleure (en terme d'inertie) que celle pour $k = 4$.

- Proposer un script permettant de déterminer quel est le nombre de clusters optimal pour le jeu de données `data2`.

```
bestcluster = []
for k in range(2, 10) :
    s = silhouette(bestKmeans(data2, k))
    bestcluster.append(s)
x = [k for k in range(2, 10)]
plt.plot(x, bestcluster)
```

On obtient la figure suivante.

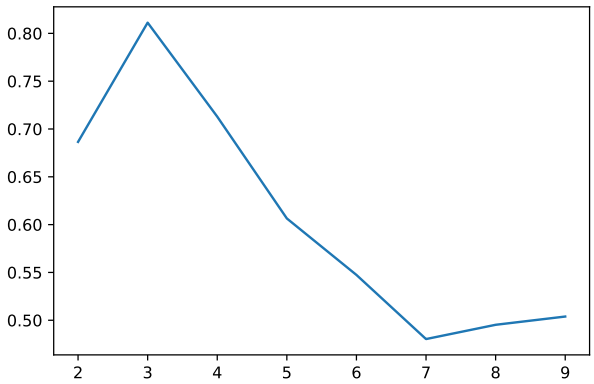

Fig. 3 Valeurs des silhouettes des partitions de `data2` en fonction de k

On constate que le nombre de 3 clusters est celui qui correspond au facteur de silhouette maximal. C'est donc le nombre qui fournit la meilleure partition (pour l'inertie).