

Chap.3 : Algorithmes d'apprentissage

Table des matières

1	Apprentissage supervisé	2
1.1	Un premier exemple	2
1.2	Reconnaissance de caractères	6
2	Apprentissage non supervisé	8
2.1	Partitionnement des données	8
2.2	Reconnaissance de caractères	10
2.3	Application à la compression de données	11

Sous le terme d'**intelligence artificielle** se cachent un ensemble de techniques permettant à des machines d'accomplir des tâches et de résoudre des problèmes normalement réservés aux humains, comme par exemple reconnaître et localiser des objets dans une image.

Depuis quelques années, on associe presque toujours l'intelligence aux capacités d'apprentissage : c'est grâce à l'apprentissage qu'un système intelligent capable d'exécuter une tâche peut améliorer ses performances avec l'expérience.

On distingue deux type d'apprentissage : le plus fréquent, l'apprentissage supervisé, consiste pour un opérateur à montrer à la machines des milliers voire des millions d'exemples étiquetés avec leur catégorie, qui permettront à la machine de déterminer elle-même les paramètres pertinents pour classer chaque objet dans la catégorie qui lui correspond.

Une fois cette phase d'apprentissage terminée, la machine doit être capable de généraliser à des objets pas encore vu.

L'apprentissage non supervisé est plus ambitieux, mais il est aussi plus proche de notre propre modèle d'apprentissage, basé sur l'observation. 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 !

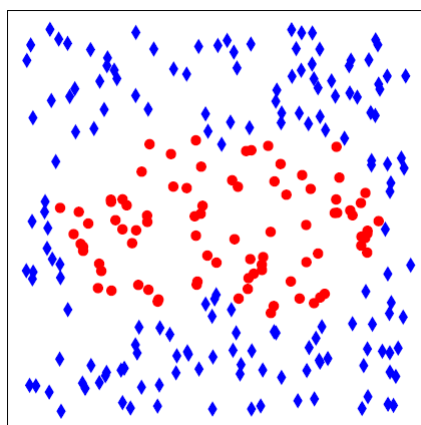
Dans ce chapitre nous allons nous intéresser à deux algorithmes que l'on utilise dans ces deux modes d'apprentissage :

- l'algorithme des k plus proches voisins
- l'algorithme des k moyennes.

1 Apprentissage supervisé

1.1 Un premier exemple

Nous allons considérer un ensemble de points de coordonnées $(x, y) \in [0, 1]^2$, qui peuvent appartenir à deux catégories représentées par des losanges bleus et des disques rouges et nous allons fournir à la machine un jeu de 250 données d'entraînement .



Une fois ces données acquises nous souhaitons qu'à partir d'un point de coordonnées $(x, y) \in [0, 1]^2$, la machine lui attribue une catégorie (bleu ou rouge).

La méthode des k plus proches voisins consiste à déterminer les k données d'entraînement les plus proches du point (x, y) , et à attribuer à ce point la catégorie majoritaire parmi ces voisins.

Dans l'algorithme qui suit, nous allons supposer que *données* est un tableau contenant les données d'apprentissage sous la forme (x, y, c) avec $(x, y) \in [0, 1]^2$ et $c = \text{"blue"}$ ou $c = \text{"red"}$.

```
def plusProchesVoisins(donnees, m, k):  
    • t = sorted(donnees, key=lambda d: (d[0] - m[0])**2 + (d[1] - m[1])**2)[:k]  
    • r = 0  
    • for d in t:  
    •     if d[2] == "r":  
    •         r += 1  
    • if 2 * r > k:  
    •     return "r"  
    • else:  
    •     return "b"
```

Afin de mieux comprendre la première instruction de cette fonction, on se référera à la documentation Python :

Key Functions

Both `list.sort()` and `sorted()` have a `key` parameter to specify a function (or other callable) to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the `key` parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

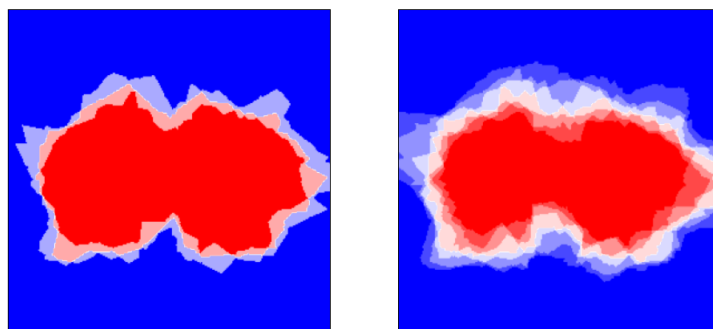
A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2]) # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Puisqu'il n'y a ici que deux catégories, si nous prenons un entier k impair nous n'aurons pas à gérer le cas d'égalité.

Avec la fonction *plusProchesVoisins*, nous disposons d'une fonction qui attribue une catégorie à tout point de $[0, 1]^2$, en se basant sur la catégorie de ces k plus proches voisins.

Voici les catégories attribuées à chacun des points par cet algorithme pour $k = 3$ et pour $k = 7$:



Application 1.1. 1. Générer un tableau de 10^4 valeurs ainsi défini :

- chaque valeurs pouvant contenir un des trois caractères 'w', 'b' ou 'r' selon que le point soit blanc, bleu ou rouge.
- 100 couleurs (autre que blanc) sont placées aléatoirement dans ce tableau, les autres valeurs correspondent à du blanc.

```

• Tableau=[]

• for x in range(100):
•     l=[]
•     for y in range(100):
•         l.append([x,y,'w'])
•     Tableau.append(l)

• donnees=[]
• cpt=0
• while cpt<100:
•     x=rd.randint(0,99)
•     y=rd.randint(0,99)
•     if Tableau[x][y][2]=='w':
•         couleur=rd.randint(1,3)
•         if couleur==1:
•             Tableau[x][y][2]='b'
•         else:
•             Tableau[x][y][2]='r'
•     donnees.append(Tableau[x][y])
•     cpt=cpt+1

```

2. $k \in \mathbb{N}$ et $(x, y) \in \llbracket 0; 99 \rrbracket^2$ un couple d'entiers sont saisis par l'utilisateur.

Renvoyer la couleur majoritaire (bleu ou rouge) parmi les k voisins de $(x; y)$.

```

• affichage(Tableau, 'Version1.png')

• k=int(input('saisir k'))
• x=int(input('saisir x'))
• y=int(input('saisir y'))
• print(plusProchesVoisins(donnees, [x,y], 3))

```

Comment interpréter les résultats que nous avons obtenus ?

Dans un problème d'apprentissage automatique, il est d'usage de séparer les données en deux sous-ensembles :

- les données réservées à l'apprentissage
- les données destinées à tester la qualité de l'apprentissage.

Nous allons donc supposer que l'on dispose d'un tableau test contenant 100 autres données, toujours sous la forme (x, y, c) avec $(x, y) \in [0, 1]^2$ et $c = \text{"blue"}$ ou $c = \text{"red"}$, et nous allons appliquer la méthode du plus proche voisin à chacun de ces points. Si notre objectif est de reconnaître si un point est rouge ou pas, les points tests vont se répartir en quatre catégories :

- les vrais positifs (les points rouges reconnus comme tels) ;
- les vrais négatifs (les points bleus reconnus comme tels) ;

- les faux positifs (les points bleus reconnus à tort comme rouges);
- les faux négatifs (les points rouges reconnus à tort comme bleus).

Ces quatre valeurs sont rangées dans une matrice de la forme :

$$C = \left(\begin{array}{c|c} \text{VP} & \text{FN} \\ \hline \text{FP} & \text{VN} \end{array} \right)$$

appelée **matrice de confusion**.

Voici par exemple les matrices que l'on obtient avec différentes valeurs de k , pour le même jeu de données et de test :

k	1	3	7	17	31
C	$\begin{pmatrix} 27 & 4 \\ 3 & 66 \end{pmatrix}$	$\begin{pmatrix} 30 & 1 \\ 3 & 66 \end{pmatrix}$	$\begin{pmatrix} 28 & 3 \\ 4 & 65 \end{pmatrix}$	$\begin{pmatrix} 27 & 4 \\ 4 & 65 \end{pmatrix}$	$\begin{pmatrix} 28 & 3 \\ 10 & 59 \end{pmatrix}$

La matrice de confusion peut aider à choisir la meilleure valeur de k ; ici par exemple on choisira plutôt $k = 3$.

Application 1.2. En reprenant les données de l'application précédente, écrire un algorithme sous Python générant la matrice de confusion pour $k \in \mathbb{N}$.

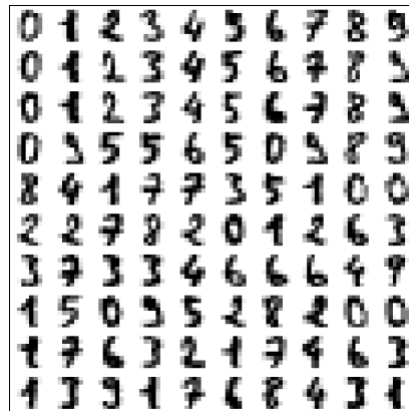
```
def matrice_confusion(tableau, test):
    n=len(test)
    for i in range(n):
        x=test[i][0]
        y=test[i][1]
        couleur=test[i][2]
        VP=0
        VN=0
        FP=0
        FN=0
        if tableau[x][y][2]==couleur and couleur=='r':
            VP+=1
        elif tableau[x][y][2]==couleur and couleur=='b':
            VN+=1
        elif tableau[x][y][2]!=couleur and couleur=='b':
            FP+=1
        else :
            FN+=1
    return(VN,VP,FN,FP)
```

1.2 Reconnaissance de caractères

Dans la réalité on n'utilise pas la totalité des échantillons pour l'apprentissage : une partie est préservée pour servir à tester la qualité de la reconnaissance.

Dans l'exemple qui suit, nous disposons de 1797 images de 8×8 pixels en niveau de gris représentant des chiffres de 0 à 9.

Voici un exemple :



Plus précisément, ces données nous sont fournies par l'intermédiaire de deux tableaux :

- X est une matrice de taille 1797×64 ; pour tout $k \in \llbracket 0, 1796 \rrbracket$, $X[k]$ est un vecteur de \mathbb{R}^{64} qui représente l'image digitalisée d'un chiffre;
- Y est un vecteur de taille 1797; pour tout $k \in \llbracket 0, 1796 \rrbracket$, $Y[k]$ est le label de $X[k]$, autrement dit l'entier compris entre 0 et 9 qui est représenté par $X[k]$.

Nous allons partager ces données en deux parties sensiblement égales :

- la première consacrée à l'apprentissage
- la seconde consacrée au test.

Dans la partie consacrée à l'apprentissage, on va choisir 90 représentants de chacune de nos dix classes, il restera donc $1797 - 90 \times 10 = 897$ données pour nous servir de test.

```
• donnees = set()
• test = set()
• nb = [0] * 10
• for i in range(1797):
•     if nb[Y[i]] < 90:
•         donnees.add(i)
•         nb[Y[i]] += 1
•     else:
•         test.add(i)
```

Soient deux images $Y1$ et $Y2$, définissons la distance entre ces deux images :

```
def dist(Y1, Y2):
    return np.linalg.norm(Y1-Y2)
```

Écrivons maintenant le fonction *plusProchesVoisins* correspondante. Cette fonction renvoie la ou les labels les plus présents parmi les k voisins (sous la forme d'une liste) de la i ème image :

```
def plusProchesVoisins(i, k):
    t = sorted(donnees, key=lambda j: dist(X[i], X[j]))[:k]
    d = [Y[j] for j in t]
    m, sol = 0, []
    for c in range(10):
        if d.count(c) > m:
            m, sol = d.count(c), [c]
        elif d.count(c) == m:
            sol.append(c)
    return sol
```





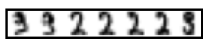
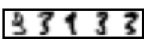

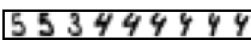

Appliqué à une donnée de test, trois cas de figure sont possibles :

- la fonction *plusProchesVoisins* renvoie une seule valeur, exacte ;
- la fonction *plusProchesVoisins* renvoie une seule valeur, inexacte (la machine se trompe) ;
- la fonction *plusProchesVoisins* renvoie plusieurs valeurs (la machine est indécise).

L'expérience montre que pour $k = 3$, sur les 897 valeurs de test :

- la machine reconnaît le bon caractère 858 fois (soit un taux de réussite de plus de 95%) ;
- la machine se trompe 31 fois ;
- la machine est indécise 8 fois.

Les erreurs que la machine a commises sont présentées ci dessous :

0		6	
1		7	
3		8	
4		9	
5			

2 Apprentissage non supervisé

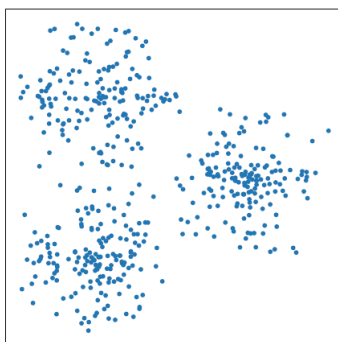
2.1 Partitionnement des données

Nous allons maintenant nous intéresser à un problème plus complexe : comment comprendre la structure des données sans aider la machine en lui

donnant des données d'apprentissage étiquetées ?

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.

Observons la figure suivante :



Nous voyons tous trois classes de points. Nous devons faire en sorte qu'il en soit de même de la machine.

Nous allons quand même l'aider un peu en lui imposant le nombre k de classes.

En appelant C_1, \dots, C_k ces classes, on note :

- μ_j le barycentre de C_j , autrement dit $\mu_j = \frac{1}{\text{card } C_j} \sum_{x \in C_j} x$;
- m_j le moment d'inertie de C_j , soit $m_j = \sum_{x \in C_j} \|x - \mu_j\|^2$.

L'algorithme des k moyennes

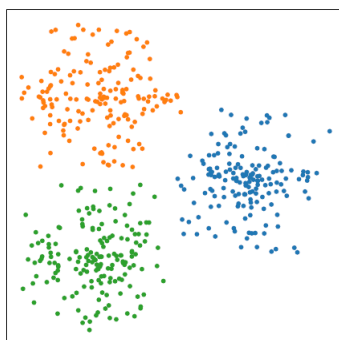
Le calcul de la classification optimale, c'est-à-dire minimisant la somme des moments d'inertie, est un problème très coûteux en temps, aussi allons nous employer un algorithme glouton. Ce dernier nous assurera d'obtenir in fine une «bonne» solution, mais pas forcément la meilleure. L'algorithme des k moyennes consiste à réaliser la succession d'opérations suivantes :

1. on choisit aléatoirement k centres μ_1, \dots, μ_k ;
2. chacun des points du nuage est associé au centre μ_j le plus proche ; on crée ainsi k classes C_1, \dots, C_k ;
3. on calcule les barycentres μ_1, \dots, μ_k de ces classes, qui remplacent les valeurs précédentes ;
4. on reprend le calcul à partir de l'étape (2).

Nous admettrons que cet algorithme converge, autrement dit qu'à partir d'une certaine étape les centres μ_1, \dots, μ_k ne se déplacent plus et donc que les classes C_1, \dots, C_k sont stabilisées.

Mais répétons-le, cet algorithme donne une configuration pour laquelle la somme des moments d'inertie est un minimum local, mais pas forcément le minimum global.

Voici le résultat de l'algorithme des k moyennes pour $k = 3$:



2.2 Reconnaissance de caractères

Considérons de nouveau 1797 images de chiffres entre 0 et 9, mais cette fois-ci sans prendre en compte leur label.

Nous allons appliquer l'algorithme des k moyennes avec $k = 10$, de manière à obtenir un regroupement de ces images en 10 classes, qu'on espère correspondre aux différents chiffres représentés.

Rappelons que ces images sont regroupées dans une liste X , chaque image $X[i]$ étant représentée par un vecteur de \mathbb{R}^{64} .

Nous commençons par écrire une fonction qui calcule le barycentre d'un ensemble d'images. Ce dernier est un vecteur de \mathbb{R}^{64} ; il représente lui aussi une image 8×8 .

```
def barycentre(s):  
    return sum([X[j] for j in s]) / len(s)
```

Voici l'algorithme des k moyennes :

```

1 def kmoyennes(X, k):
2     mu = np.array([X[rd.randint(len(X)) for _ in range(k)])
3     while True:
4         s = [set() for _ in range(k)]
5         for i in range(len(X)):
6             dmin = np.inf
7             for j in range(k):
8                 if dist(X[i], mu[j]) < dmin:
9                     jmin, dmin = j, dist(X[i], mu[j])
10            s[jmin].add(i)
11            newmu = np.array([barycentre(s[j]) for j in range(k)])
12            if (newmu == mu).all():
13                break
14            mu = newmu
15    return s

```

Description de cet algorithme :

- La ligne 2 choisit k images initiales.
- ligne 5 : pour chaque image X_i on calcule la valeur de μ_j la plus proche de X_i (lignes 6-9) pour ranger ce dernier dans la classe associée (ligne 10).
- On recalcule les différents barycentres (ligne 11) et on s'arrête lorsque ces derniers n'ont pas été modifiés (lignes 12-13).

Voici les images correspondants aux barycentres des dix classes que l'on obtient par cet algorithme pour $k = 10$:



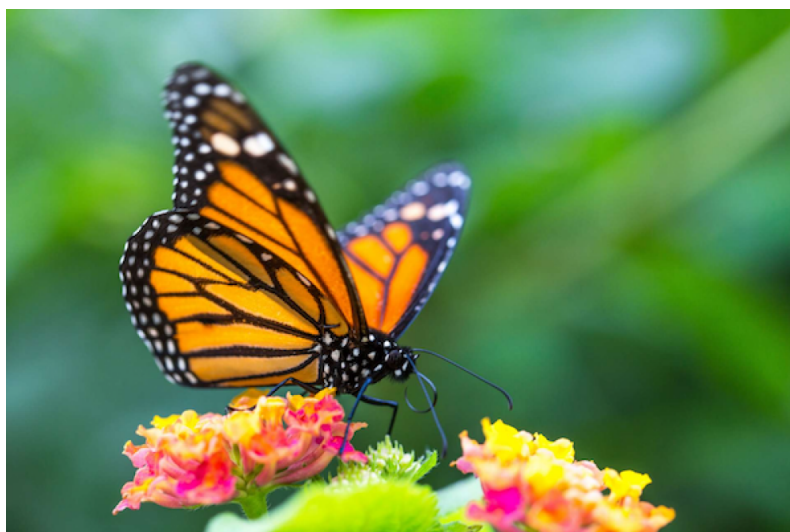
Manifestement, les classes obtenues correspondent bien à notre souhait. Si on passe maintenant en revue toutes les images de chacune de ces dix classes en tenant compte de leur label, on obtient 1426 succès et 371 échecs. Autrement dit, 79% des images ont été rangées dans les bonnes classes. Plus précisément, on peut pour chacune des 10 classes calculer le pourcentage de réussite :

classe	0	1	5	7	8	4	3	2	9	6
réussite	99%	60%	91%	85%	45%	98%	87%	85%	56%	97%

Le plus mauvais score est obtenu pour la cinquième classe qui ne comporte que 45 % d'images représentant le chiffre 8 ; à l'inverse, la première classe comporte 99% d'images représentant le chiffre 0.

2.3 Application à la compression de données

Considérons l'image suivante :



Chaque pixel est un triplet (r, g, b) où r, g et b sont des flottants de l'intervalle $[0; 1[$ codés sur 32 bits représentant la quantité de rouge, vert et bleu du pixel. L'image est représentée en machine par un tableau numpy `img` :

```
In [1]: img.shape
Out[1]: (415, 626, 3)
```

Exercice 2.1. Rédiger un script Python qui calcule le nombre de couleurs différentes utilisées dans cette image.

[illegible]

Notre objectif est de réduire ce nombre à seulement 16 couleurs. Pour ce faire, nous allons :

1. appliquer l'algorithme des k moyennes pour regrouper les différents pixels en 16 classes
2. calculer la couleur moyenne de chacune de ces 16 classes

3. attribuer cette valeur moyenne à chacun des pixels de la classe correspondante.

Exercice 2.2. 1. Rédiger une fonction **dist** (p, q) qui prend pour arguments deux pixels p et q (représentés par deux vecteurs de \mathbb{R}^3) et renvoie la distance euclidienne entre ces deux vecteurs.

2. Rédiger une fonction **initialise**(img, k) qui prend pour arguments une image img un entier k et renvoie un tableau numpy de k cases, chacune d'elles contenant un pixel tiré au hasard dans l'image.
3. Rédiger une fonction **barycentre**(img, s) qui prend pour argument une image img et un ensemble s de coordonnées (x, y) et renvoie un pixel égal au barycentre des pixels de l'image dont les coordonnées appartiennent à s .
4. Rédiger une fonction **plusProchePixel** (p, μ) qui prend pour argument un pixel p et une liste de pixels $[\mu_0, \dots, \mu_{k-1}]$ et qui renvoie l'indice j qui minimise la distance $\|p - \mu_j\|$.
5. En déduire une fonction **kmoyennes** (img, k) qui prend pour arguments une image et un entier k et qui renvoie un tableau s de longueur k , chacun de ses éléments étant un ensemble de coordonnées des pixels obtenu par l'algorithme des k moyennes.

Exercice 2.3. Rédiger une fonction **reduire** (img, k) qui prend pour argument une image et renvoie une nouvelle image dans laquelle seules k couleurs sont utilisées.

Voici l'image après réduction à 16 couleurs :

