Chapitre V
Algorithmes d'apprentissage

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 vus.
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, et 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'entrainement (voir figure 5.1).

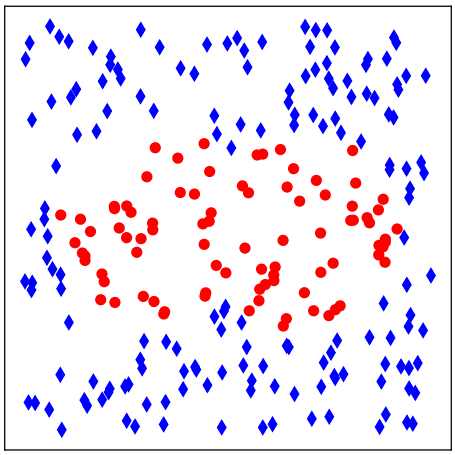

Figure 5.1 – 250 données d'entrainement.

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'entrainement 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 `donnees` est un tableau contenant les données d'apprentissage sous la forme (x,y,c) avec $(x,y) \in [0,1]^2$ et $c =$ *"blue"* ou $c =$ *"red"*.

```
def plusProchesVoisins(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] == "red":
            r += 1
    if 2 * r > k:
        return "red"
    else:
        return "blue"
```

Remarque. La fonction `sorted` possède un argument facultatif `key` qui indique, le cas échéant, une fonction qui est appelée sur chaque élément de la liste avant d'effectuer les comparaisons.
Dans le cas présent, chaque donnée d est triée suivant les valeurs croissantes de $(d_0-m_0)^2+(d_1-m_1)^2$, autrement dit suivant le carré de la distance euclidienne entre les points d et m.

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 ses k plus proches voisins. J'ai représenté figure 5.2 les catégories attribuées à chacun des points du plan par cet algorithme pour $k=3$ et pour $k=7$.

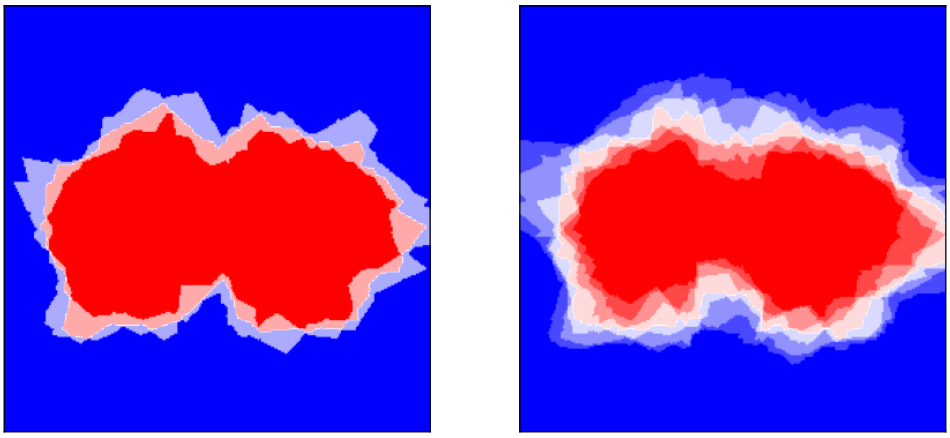

FIGURE 5.2 – Le résultat de `plusProchesVoisins` pour $k=3$ et $k=7$.

Matrice de confusion

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 et 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 $c=$ *"blue"* ou $c=$ *"red"*, et nous allons appliquer la méthode du plus proche voisin à chacun de ces points. Si notre objectif est de reconnaitre 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 $k = 3$.

1.2 Reconnaissance de caractères

Dans la réalité on 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 1 797 images de 8×8 pixels en niveau de gris représentant des chiffres de 0 à 9 (un extrait de la base de données est présenté figure 5.3).

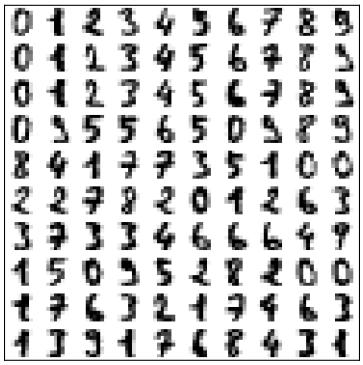

Figure 5.3 – Un extrait de la banque de données.

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

- `X` est une matrice de taille $1\,797 \times 64$; pour tout $k \in [\![0, 1796]\!]$, `X[k]` est un vecteur de $\mathbb{R}^{64}$ qui représente l'image digitalisée d'un chiffre ;
- `Y` est un vecteur de taille 1 797 ; pour tout $k \in [\![0, 1796]\!]$, `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 au test. Dans la partie consacrée à l'apprentissage, on va choisir 90 représentants de chacune de nos dix classes, il restera donc 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)
```

Définissons la distance entre les images i et j :

```
def dist(X, Y):
    s = 0
    for i in range(len(X)):
        s += (X[i] - Y[i]) ** 2
    return s
```

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).

```
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 reconnait 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 à commise sont présentées figure 5.4.

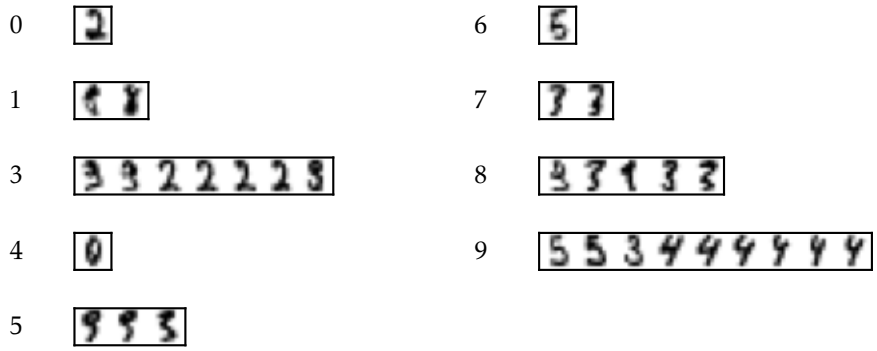

Figure 5.4 – Les erreurs commises par la machine.

Quelles améliorations envisager ?

La méthode que je viens de vous présenter est bien entendu simpliste, mais peut être améliorée de plusieurs façons, par exemple :

- au lieu de considérer les k plus proches voisins, considérer tous les voisins contenus dans une boule de rayon ϵ centrée autour de l'objet à classer ;
- pondérer la contribution de chaque voisin par un coefficient inversement proportionnel à la distance à l'objet à classer ;
- ou encore utiliser d'autres distances que la distance euclidienne.

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.

Par exemple, en regardant la figure 5.5 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.

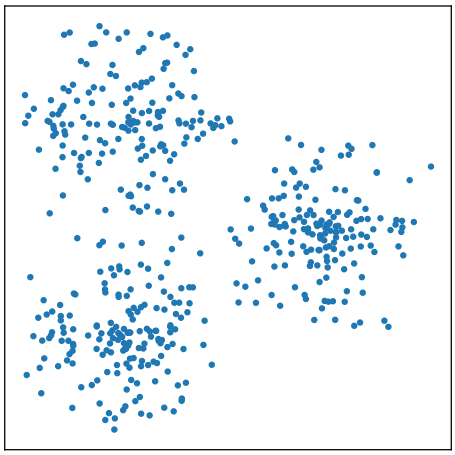

FIGURE 5.5 – Les données semblent se regrouper en trois classes.

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

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

L'objectif de la machine va être de minimiser la somme des moments d'inertie $\displaystyle\sum_{j=1}^{k} m_j$.

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 $\mu_1, \dots, \mu_k$;
(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 $\mu_1, \dots, \mu_k$ de ces classes, qui remplacent les valeurs précédentes ;
(4) on reprend le calcul à partir de l'étape (2).

Nouas admettrons que cet algorithme converge, autrement dit qu'à partir d'une certaine étape les centres $\mu_1, \dots, \mu_k$ 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.

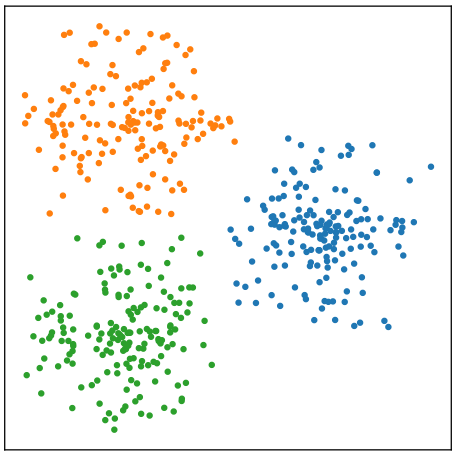

FIGURE 5.6 – Le résultat de l'algorithme des k moyennes pour $k = 3$.

2.2 Reconnaissance de caractères

Considérons de nouveau nos 1 797 images de chiffres de 0 à 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)
```

Passons maintenant à 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
```

La ligne 2 choisit k images initiales. Ensuite (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 $k = 10$:

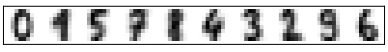

FIGURE 5.7 – Les barycentres des 10 classes obtenues.

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 1 426 succès et 371 échecs. Autrement dit, 79 % des images ont étés 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 représentée figure 5.8. Ses dimensions sont de 415×626 pixels. Chaque pixel est un triplet

FIGURE 5.8 – L'image initiale et sa réduction à 16 couleurs.

(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)
```

Cette image initiale comporte 121 915 couleurs différentes. Notre objectif est de réduire ce nombre à seulement 16 couleurs. Pour ce faire, on applique l'algorithme des k moyennes pour regrouper les différents pixels en 16 classes de couleurs voisines. Pour ce faire, il faut définir une distance entre deux couleurs c et c', mais puisque c et c' sont représentées par un triplet de $[0, 1]^3$, on peut utiliser une distance euclidienne :

$$\text{si } c = (r, v, b) \text{ et } c' = (r', v', b') \text{ alors } \mathrm{d}(c, c') = \sqrt{(r - r')^2 + (v - v')^2 + (b - b')^2}.$$

Une fois le regroupement en 16 classes effectué, on calcule la couleur moyenne de chacune de ces 16 classes, puis on attribue cette valeur moyenne à chacun des pixels de la classe correspondante. Le résultat est une nouvelle image, présentée à droite figure 5.8.
Enfin, la figure 5.9 présente deux réductions à 16 couleurs, la première avec la norme euclidienne, la seconde avec la norme infinie.

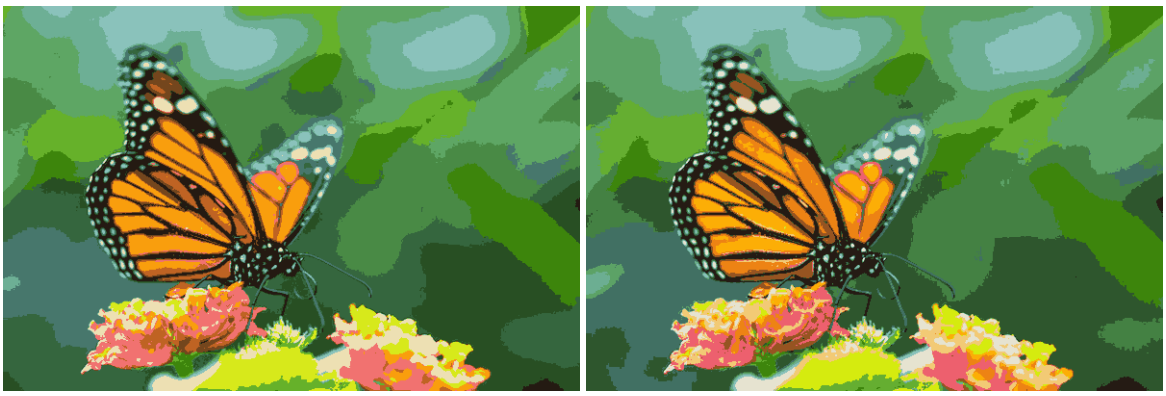

FIGURE 5.9 – les réductions obtenues avec la norme euclidienne et la norme infinie.