Chapitre 6
Algorithmes d'apprentissage (intelligence artificielle)

Le terme Intelligence Artificielle étant assez galvaudé et probablement mal compris et mal utilisé, nous utiliserons dans ce chapitre le terme d'algorithme d'apprentissage où nous essaierons effectivement de faire apprendre quelque chose à l'ordinateur. Dans ce chapitre nous allons voir deux algorithmes d'apprentissages. Il s'agit pour l'ordinateur de regrouper des données par similitude et y coller une étiquette qui ait du sens pour les humains. Si distinguer des chats des chiens est assez simple pour un humain, il n'en va pas de même pour un ordinateur qui ignore tout de ces concepts animaliers.

Table des matières

1 Présentation des données

Nous allons utiliser les données issues d'une bibliothèque Python nommée `sklearn` dédié à l'intelligence artificielle. Avec les lignes :

```
from sklearn import datasets

digits=datasets.load_digits()
Images=[digits.images[k].tolist() for k in range(len(digits.images))]
Numeros=list(digits.target)
```

nous disposons d'une liste d'images nommée `Images`. Chaque image représente un chiffre manuscrit. De plus, pour chaque image, le chiffre manuscrit a été indiqué manuellement dans la liste `Numeros`. Par exemple, si on fait :

```
plt.figure()
plt.imshow(Images[13], cmap='binary')#Affichage de l'image d'indice 13
plt.title(Numeros[13])#titre de l'image: son numéro
plt.show()
```

On obtient :

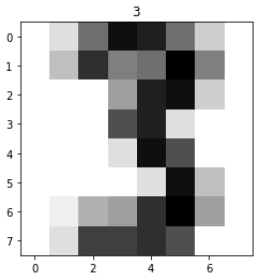

FIGURE 1 – Une image et son étiquette : le chiffre sur l'image indiqué par un humain.

2 L'algorithme des k plus proches voisins

Supposons que l'on ait des données avec N éléments classifiés par une étiquette. Ici l'étiquette est le chiffre représenté par l'image. On souhaite deviner l'étiquette d'un élément n'appartenant pas à l'ensemble des données.

Pour cela on suppose que l'on a une notion de distance qui mesure la proximité entre deux éléments. Par exemple :

- Si on a deux nombres réels ou complexes x et y, alors $|x-y|$ représente la distance entre x et y.
- Si on a $x=(x_1,x_2,\dots,x_n)\in\mathbb{R}^n$ et $y=(y_1,y_2,\dots,y_n)\in\mathbb{R}^n$, alors $d(x,y)=\sqrt{\langle x-y,x-y\rangle}$ où $\langle x,y\rangle=\sqrt{\sum_{k=1}^{n}x_ky_k}$.
- Si on a deux fonctions $f\in\mathscr{C}^0([a\,;b],\mathbb{R})$ et $g\in\mathscr{C}^0([a\,;b],\mathbb{R})$, alors $d(f,g)=\sqrt{\langle f-g,f-g\rangle}$, où $\langle f,g\rangle=\int_a^b f(x)g(x)\,\mathrm{d}x$ mais $d'(f,g)=\|f-g\|_\infty$ est une autre possibilité.
- Si on a deux matrices $A\in\mathscr{M}_{n,p}(\mathbb{R})$ et $B\in\mathscr{M}_{n,p}(\mathbb{R})$, alors $d(A,B)=\sqrt{\langle A-B,A-B\rangle}$ avec $\langle A,B\rangle=\mathrm{tr}(A^TB)=\sum_{i=1}^{n}\sum_{j=1}^{p}a_{i,j}b_{i,j}$

C'est cette dernière méthode que nous allons utiliser car ici, nous avons des images assimilées à des matrices carrées de taille 8. Nous pouvons donc coder la fonction distance par :

```
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 1. D'une manière générale, $d(x,y)=N(x-y)$ définit toujours une distance dès que N est une norme (cf. cours de maths).

De là, on a une première méthode qui va permettre de classifier notre élément. On va fixer $k \in \mathbb{N}^*$. Et on va regarder les k éléments de nos données les plus proches de notre élément (suivant la distance que l'on s'est fixée) et on prendra l'étiquette majoritaire.

Pour cela, on peut commencer par écrire une fonction qui dans une liste `L` va renvoyer l'élément (ou un élément) dont le nombre d'occurrences dans `L` est maximal.

```
def Majoritaire(L):
    """Renvoie un élément de L dont le nombre d'occurrences est maximal"""
    D={}# Dictionnaire du nombre d'occurrences des éléments de L
    Max=0
    for e in L:# Pour chaque élément de la liste
        if e in D:#Si on a déjà rencontré cet élément
            D[e]=D[e]+1#Alors on l'a rencontré une fois de plus
        else:
            D[e]=1#Sinon c'est la première fois qu'on le rencontre et donc on l'a rencontré une fois
        if D[e]>=Max:#S'il dépasse Max
            Max,emax=D[e],e#On actualise Max et emax l'élément qui a été rencontré Max-fois
    return emax
```

Nous pouvons donc coder l'algorithme appelé k plus proches voisins, qui étant donné une image non étiquetée retournera l'étiquette majoritaire parmi les k images les plus proches de notre image test.

Pour cela, faisons un rappel sur les tris de liste. Si `L` est une liste de nombres, on dit qu'elle est triée (par ordre croissant) si pour tout `i` entre 0 et `len(L)-2`, `L[i]`$\leqslant$`L[i+1]`. Seulement ici, on a des images que l'on veut trier par proximité à une autre image. On a donc besoin de la notion de clé de tri. Une clé de tri est une fonction d'un ensemble `E` dans $\mathbb{R}$. Et on dit qu'une liste `L` d'éléments de `E`, on dit qu'elle est triée (par ordre croissant) suivant la clé de tri `h` si pour tout `i` entre 0 et `len(L)-2`, `h(L[i])`$\leqslant$`h(L[i+1])`.

```
def PlusProchesVoisins(i,k):
    """Renvoie l'étiquette majoritaire parmi les k images les plus proches de images[i]"""
    def cle(j):#Clé de tri
        return Dist(Images[i],Images[j])
    trié=sorted(donnees,key=cle)#la liste données est trié avec la clé:
        #les premiers éléments de trié sont les plus proches de images[i]
    L=[Numeros[j] for j in trié[:k]]
    return Majoritaire(L)
```

Pour tester notre algorithme, il va falloir deux sortes d'éléments : nos données qui sont les éléments qui vont faire apprendre la correspondance éléments->étiquette et les éléments tests que l'on va tester pour comparer la prévision de l'algorithme à l'étiquette que l'on connaît.

```
N=900
donnees=list(range(N))#données pour l'algorithme
test=list(range(N,len(Images)))#les images que l'on va utiliser pour tester la réponse de l'algorithme.
```

Ici, en fait toutes les images qui sont dans `test` sont toutes étiquetées, ainsi si l'algorithme donne un résultat, on peut le comparer à l'étiquette pour savoir si l'algorithme à bien deviné. On note $C_{i,j}$ le nombre de fois où une image portant l'étiquette i, l'algorithme lui a attribué une étiquette j. On obtient alors une matrice. Plus cette matrice est diagonale, plus cela veut dire que l'algorithme a bien deviné. Cette matrice s'appelle **matrice de confusion** que l'on peut programmer :

```
def MatriceDeConfusion(k):
    C=[[0 for j in range(10)] for i in range(10)]
    S=0#Nombre de succès (nb de prédictions justes de l'ordinateur)
    for l in test:
        i=digits.target[l]#le vrai chiffre sur Images[i]
        j=PlusProchesVoisins(l,k)#la prédiction donné par l'algorithme k-PPV
        C[i][j]=C[i][j]+1
        if i==j:#bonne prédiction
            S=S+1   #On compte un succès de plus
    for i in range(10):
        print(C[i])#on affiche la liste
    print("Pourcentage de réussite:",S*100/len(test))
```

On dit que l'algorithme des k plus proches voisins est un algorithme supervisé, en effet, il fonctionne en ayant donné à l'ordinateur les étiquettes des données pour qu'il puisse s'en servir sur les images tests.

3 L'algorithme des k-moyennes

Nous allons maintenant voir un algorithme **non supervisée**, c'est-à-dire que nous n'allons pas dire à l'ordinateur quel chiffre est sur l'image. Nous allons lui demandé de séparer les données en k-groupes par proximité. L'entier k, qui correspond au nombre de groupes sera imposé par l'utilisateur.

Pour cela, partons du principe que l'on ait ces k-groupes, alors on peut calculer le barycentre de chacun de ses groupes. Réciproquement, si on avait seulement les barycentres de chacun de ses groupes, on pourrait reconstituer les groupes (appelés aussi 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 classes. Et on note $b_i = \dfrac{1}{|C_i|}\sum\limits_{x\in C_i} x$ le barycentre de C_1 et m_i le moment d'inertie de C_i soit $m_i = \sum\limits_{x\in C_i} \|x-b_i\|^2$.

```
def Barycentre(cluster):
    B=[[0 for j in range(8)] for i in range(8)]#image vide au déaprt
    for i in range(8):
        for j in range(8):
            for k in cluster:
                B[i][j]=B[i][j]+Images[k][i][j]
            B[i][j]=B[i][j]/len(cluster)
    return B
```

L'idéal serait d'obtenir des classes tels que $\sum\limits_{i=1}^{k} m_i$ soit minimal. Mais c'est un problème de minimisation difficile à réaliser.

Nous allons procéder de la façon suivante : nous allons choisir k barycentres de façon aléatoire parmi nos données. Puis nous allons répéter ces deux étapes jusqu'à convergence :

- 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 :

```
def PlusProcheBarycentre(B,I):
    """Renvoie le barycentre de B le plus proche de l'image 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
```

A l'aide d'une boucle `while` nous allons réaliser ces deux étapes. Nous allons comparer l'ancienne liste des barycentres à la nouelle et arrêter la fonction dès qu'il n'y a plus de changements.

```
def kMoyennes(data,k):
    B=[data[np.random.randint(len(data))] for i in range(k)]
    while 0==0:
        Clusters=[[] for i in range(k)]
        for i in range(len(data)):
            jmin=PlusProcheBarycentre(B,data[i])
            Clusters[jmin].append(i)
        nouveauB=[Barycentre(Clusters[j]) for j in range(k)]
        if nouveauB==B:
            return Clusters#Arrêt de la fonction
        B=nouveauB
```

Nous allons admettre que cet algorithme termine bien. C'est-à-dire qu'il existe un moment où les nouveaux barycentres sont identiques aux anciens. Cependant, cet algorithme ne converge pas vers un minimum global mais seulement vers un

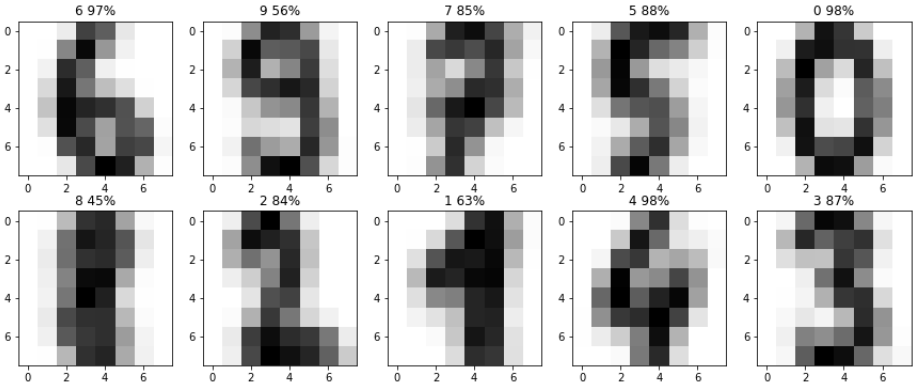

FIGURE 2 – Les 10 barycentres des 10 clusters ainsi que l'étiquette majoritaire pour chaque cluster et le pourcentage de cette étiquette majoritaire.

minimum local. D'ailleurs, on obtient des résultats différents à chaque fois que l'on teste cet algorithme. Ceci traduit la dépendance dans l'aléatoire des barycentres de départ.

Ici nous avons un algorithme dit **non supervisé**, nous n'avons pas donné les étiquettes à nos données. Et notre programme a regroupé les données en 10 paquets qui correspondent plus ou moins à des groupes de même chiffre.