Algorithmes d'apprentissage

I. Introduction

L'intelligence artificielle est un concept assez flou ; on peut dire qu'il s'agit de l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine.

Un des branches de l'I.A est l'apprentissage automatique (ou *machine learning*). Il s'agit d'un ensemble de techniques mathématiques et statistiques pour donner à l'ordinateur la capacité « d'apprendre » à partir d'un ensemble, souvent très vaste, de données.

On distingue deux type d'apprentissage : le plus fréquent, *l'apprentissage supervisé*, consiste à fournir à la machine 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 les classer en créant 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*.

Rem : je cite le programme officiel :

> La connaissance dans le détail des algorithmes de cette section n'est pas un attendu du programme. Les étudiants acquièrent une familiarité avec les idées sous-jacentes qu'ils peuvent réinvestir dans des situations où les modélisations et les recommandations d'implémentation sont guidées.

II. Apprentissage supervisé

II.1. Introduction

Dans l'apprentissage supervisé, on dispose d'une ensemble *d'observations* $D = \{X_0, \dots, X_{n-1}\}$ où pour tout i, $X_i \in E = \mathbb{R}^d$ (d est le nombre de données retenues). À chaque observation X_i est associée une *étiquette* Y_i qui appartient à un certain ensemble F.

Le but est de pouvoir associer une étiquette $Y \in F$ à un nouveau vecteur $X \in E$.

Il y a deux cas de figures :

- si F est fini, c'est un problème de *classification* (exemple : reconnaissance de caractères, reconnaissance de fleurs, reconnaissance d'espèces animales etc...)
- si F est infini (généralement une partie de $\mathbb{R}$ ou d'un espace vectoriel), c'est un problème de *régression* (exemple : prédire le poids, l'âge...).

II.2. Utilisation de Python

Pour illustrer ce chapitre, nous utiliserons un fichier qui permet de classifier les bouquetins. Dans ce fichier, il y a 3 espèces de bouquetins : le bouquetin des Alpes (ibex), le bouquetin ibérique (pyrenaica) et la chèvre du Caucase (caucasia)
Chaque ligne du fichier contient les valeurs des champs suivants : espèce, hauteur au garrot en cm, longueur museau-queue, longueur des cornes, masse (en kg), sexe, altitude minimale d'habitat (en m), altitude maximale d'habitat. Exemple :

```
espèce,hauteur_au_garrot_cm,longueur_museau_queue_cm,longueur_cornes_cm,masse_kg,sexe,habitat_min_m,hab
ibex,85.0,149.0,92.0,90.0,mâle,500,3300
ibex,80.0,103.0,23.0,46.0,femelle,500,3300
....
pyrenaica,64.0,102.0,19.0,36.0,femelle,500,2500
pyrenaica,63.0,137.0,80.0,90.0,mâle,500,2500
...
caucasia,102.0,167.0,70.0,79.0,mâle,800,4000
caucasia,86.0,135.0,24.0,61.0,femelle,800,4000
```

Nous récupérons d'abord dans une liste X les données (on se limitera pour l'instant aux 3 premiers champs de longueurs) et dans une liste Y les étiquettes ; par exemple on aura :

`X[0] = [85.0, 149.0, 92.0]` et `Y[0] = 'ibex'`

Cela se fait facilement avec le petit script ci-dessous.

```
def LectureBase():
    Fichier = open("data-bouquetins.txt", "r")
    contenu = Fichier.read()
    Lignes = contenu.split('\n')
    Lignes.pop(0)
    X = []
    Y = []
    for ligne in Lignes:
        champs = ligne.split(",")
        Y.append(champs[0])
        X.append( [float(champs[1]), float(champs[2]), float(champs[3])] )
    return X, Y
```

II.3. Algorithme des k plus proches voisins (KNN)

Il s'agit d'un algorithme de classification très simple. L'idée est la suivante :

- On définit sur E une distance (dite *distance de similarité*) telle que plus deux éléments sont « ressemblants », plus leur distance est petite. Pour $E = \mathbb{R}^d$, on peut utiliser la distance euclidienne.
- Pour prédire l'étiquette d'un élément X, on cherche alors dans l'ensemble D ses k plus proches voisins au sens de cette distance. Le choix de k est délicat (ni trop petit, ni trop grand).
- Le résultat de la prédiction sera l'étiquette majoritaire dans l'ensemble précédent (pour éviter les ex-aequo, on prend en général k impair).

Voici une implémentation possible en Python.

```
def dist(X1, X2):
    # distance entre deux observations
    # ici, c'est la distance euclidiennne au carré
    return sum( [ (X1[i] - X2[i])**2 for i in range(len(X1)) ] )

def KNN(X, Y, z, k):
    # liste des distances et numéro de la donnée associée
    D = [ [dist(z, X[i]), i] for i in range(len(X)) ]
    #on trie et on retient les k plus petits
    PlusProches = sorted(D, key = lambda L: L[0])[:k]
    voisins = [ p[1] for p in PlusProches ]
    # on cherche le voisin majoritaire
    voisins_uniques = list(set(voisins))
    max = 0
    for v in voisins_uniques:
        nb_occ = voisins.count(v)
        if nb_occ > max:
            max = nb_occ
            sol = v
    return Y[sol]
```

II.4. Évaluation des résultats

Pour tester la qualité de l'apprentissage, on va partager l'ensemble des données en deux parties : celles destinées à l'apprentissage (*jeu d'entraînement*) et celles destinées à tester la qualité de notre algorithme et à trouver la meilleure valeur de k (*jeu de test*).

EXERCICE 1

1. Écrire la fonction `ConstructionJeuDeTest(X, Y)` qui prend en paramètre les données X et Y et renvoie un jeu d'entrainement $X_{\text{train}}, Y_{\text{train}}$ et un jeu de test $X_{\text{test}}, Y_{\text{test}}$; pour former le jeu de test, on prendra une ligne sur 3 du jeu complet.
2. Modifier la fonction précédente afin que le jeu de test représente une proportion `pourcentage` des données et de sorte que la proportion de chaque espèce dans ce jeu soit la même que dans l'ensemble total. On utilisera la fonction `random.sample(L, n)` qui permet de choisir au hasard n élements dans une liste L.

Une fois déterminé le jeu de test et le jeu d'entrainement, on va pouvoir obtenir la *matrice de confusion*. Il s'agit d'une matrice M de $\mathcal{M}_c(\mathbb{N})$, où c est le nombre d'étiquettes, telle que, pour tout $(i,j) \in [\![0\,;c-1]\!]^2$, $M_{i,j}$ est le nombre de données de test qui ont réellement i pour étiquette et auxquelles notre algorithme KNN a attribué l'étiquette j.

Le taux de réussite est donc le nombre de cas où la réponse a été correcte, c'est-à-dire les valeurs sur la diagonale de la matrice, divisé par le nombre total de cas. On peut tracer un graphique donnant ce taux en fonction de k.

EXERCICE 2

1. Écrire la fonction `TauxReussite(M)` qui calcule le taux de réussite à partir de la matrice de confusion M.
2. Pour $k \in [\![2\,;40]\!]$ (par exemple), tracer le graphique donnant le taux de réussite en fonction de k. On pourra aussi comparer la qualité des deux jeux de test construits précédemment.

II.5. Un problème de régression

On cherche ici à estimer le poids d'un animal à partir des 3 longueurs qui le caractérisent.
Dans ce cas, l'étiquette de l'animal sera son poids. Les listes de données seront donc modifiées comme suit :

```
def ConstructionJeuDeTest1(X, Y):
    # construction d'un  jeu de test où l'on
    # prend seulement une ligne sur 3
    X_train, Y_train = [], []
    X_test, Y_test = [], []
    for i in range(len(X)):
        etiquette = Y[i]
        if i % 3 == 0:
            X_test.append(X[i])
            Y_test.append(etiquette)
        else:
            X_train.append(X[i])
            Y_train.append(etiquette)
    return X_train, Y_train, X_test, Y_test

def ConstructionJeuDeTest2(X, Y, pourcentage):
    # construction d'un  jeu de test où l'on respecte
    # les proportions de chaque catégorie
    etiquettes_diff = list(set(Y))
    # on cherche  les numéros de tous les objets
    # ayant la même étiquette
    dico_complet={ x:[] for x in etiquettes_diff }
    # et pour chacune des catégories, on en extrait proportion
    dico_test = { x:[] for x in etiquettes_diff }
    for i in range(len(X)):
        dico_complet[Y[i]].append(i)
    for x in dico_complet:
        n = len(dico_complet[x])
        dico_test[x] = random.sample( dico_complet[x], floor(n*pourcentage) )
    # on recrée les données d'entrainement et de test
```

```
    X_train, Y_train = [], []
    X_test, Y_test = [], []
    for i in range(len(X)):
        etiquette = Y[i]
        if i in dico_test[etiquette]:
            X_test.append(X[i])
            Y_test.append(etiquette)
        else:
            X_train.append(X[i])
            Y_train.append(etiquette)
    return X_train, Y_train, X_test, Y_test
```

EXERCICE 3

1. Écrire une fonction `KNN_Regression(X, Y, z, k)` qui renvoie, comme valeur estimée du poids de l'animal de données z, la moyenne des poids des k plus proches voisins.
2. Créer un jeu de test où l'on choisit simplement au hasard une certaine proportion du jeu complet.
3. Pour $k \in [\![2\,;40]\!]$ (par exemple), tracer le graphique donnant la moyenne des erreurs relatives commises en fonction de k, lorsqu'on compare le jeu de test aux résultats fournis par la fonction précédente.
4. Modifier votre programme pour prendre aussi en compte le sexe des animaux.

III. Apprentissage non supervisé

Dans l'apprentissage non supervisé, les observations $D = \{X_0, \dots, X_{n-1}\}$ ne sont plus étiquetées : c'est à l'algorithme de les répartir dans des classes pertinentes.

L'*algorithme des k-moyennes* utilise pour cela une méthode semblable à celle des algorithmes gloutons, en construisant au fur et à mesure les classes cherchées jusqu'à obtenir une solution convenable (mais ce ne sera peut-être pas la meilleure).

Le problème est donc de partitionner l'ensemble $D = \{X_0, \dots, X_{n-1}\}$ de points de $\mathbb{R}^d$ en k groupes, appelés aussi *clusters* : $D = C_0 \sqcup \dots \sqcup C_{k-1}$, de façon à minimiser la variance des poins de chaque cluster ; plus précisément, pour chaque cluster C_j, on détermine son isobarycentre G_j (dont les coordonnées sont les moyennes des coordonnées des points de C_j), et on cherche à minimiser la quantité $\displaystyle\sum_{j=0}^{k-1} \sum_{M \in C_j} d(M, G_j)^2$.

Il n'est pas envisageable de tester toutes les partitions possibles de D en k sous-ensembles, on va donc résoudre le problème de manière approchée.

L'algorithme des k-moyennes est le suivant :

1) Il y a deux façons d'initialiser le processus.
   - *Méthode de Forgy* On choisit aléatoirement k éléments de D, $G_0, \dots G_{k-1}$.
   - Méthode du partitionnement aléatoire : On partage aléatoirement l'ensemble des données en k parties à peu près égales C_j , puis on calcule l'isobarycentre G_j de chaque cluster.
2) Pour chaque X_i dans D, on cherche le minimum des distances de X_i aux G_j ; s'il est obtenu pour un indice j_0, on place X_i dans la classe de G_{j_0} ;
3) On a ainsi obtenu une première partition ; on (re) calcule alors l'isobarycentre G_j de chaque cluster ;
4) et on recommence jusqu'à ce que les clusters soient stables, c'est-à-dire ne changent pas, ce qui revient à dire que les isobarycentres ne changent pas..

Nous allons utiliser cet algorithme pour classer nos bouquetins selon l'espèce et le sexe (6 classes) d'après leurs mensurations et leur masse. Ici, chaque élément du tableau X contiendra 4 valeurs numériques (mensurations + masse), et chaque ligne du tableau Y contiendra une liste avec les deux items espèce et sexe. Cette liste Y ne servira qu'à vérifier les résultats puisque c'est le programme qui doit faire la classification.

Cela est fait par le petit programme ci-dessous.

```python
def MatriceConfusion(Y_test, Y_result):
    etiquettes_diff = list(set(Y_test))
    nb_etiquettes = len(etiquettes_diff)
    # on numérote les étiquettes pour la matrice à l'aide d'un dico
    Noms_numéros_etiquettes = {etiquettes_diff[i]:i for i in range(len(etiquettes_diff))}
    M = np.zeros( (nb_etiquettes, nb_etiquettes), dtype = int)
    for i in range(len(Y_test)):
        M[Noms_numéros_etiquettes[Y_test[i]], Noms_numéros_etiquettes[Y_result[i]] ]+= 1
    return M

def TauxReussite(M):
    return np.trace(M)/np.sum(M)

# Ensemble des données
X_complet, Y_complet = LectureBase()

# Construction des jeux de tes et entrainement
X_train, Y_train, X_test, Y_test = ConstructionJeuDeTest2(X_complet, Y_complet, 0.33)

# résultats de l'algorithme pour le jeu de test
k = 7
Y_result = []
for z in X_test:
    Y_result.append(KNN(X_train, Y_train, z, k))
M = MatriceConfusion(Y_test, Y_result)
print(M)
print('Taux de réussite:', TauxReussite(M))

# maintenant on trace le taux de réussite selon k
K = list(range(2,40))
T = []
for k in K:
    Y_result = []
    for z in X_test:
        Y_result.append(KNN(X_train, Y_train, z, k))
    T.append(TauxReussite(MatriceConfusion(Y_test, Y_result)))

# graphique sommaire!
plt.plot(K, T)
plt.show()
```

Pour implémenter l'algorithme, nous utiliserons plusieurs petits sous-programmes. On dispose toujours de la fonction `distance` vue dans le paragraphe précédent. Les clusters seront représentés par une liste `C` de longueur k, chaque élément d'indice j de cette liste étant une liste formée des points X_i qui appartiennent au cluster C_j.
On commence par la phase d'initialisation.

EXERCICE 4

1. Écrire une fonction `Forgy(X, k)` qui initialise la liste `G` des isobarycentres selon la méthode de Forgy.
2. Écrire une fonction `Isobarycentre(L)` qui étant donnée une liste de points renvoie son isobarycentre.
3. Écrire une fonction `ClustersInit(X, k)` qui initialise la liste `G` des isobarycentres selon la méthode du partitionnement aléatoire.

On pourra utiliser la fonction `random.sample(L, n)` qui extrait au hasard n éléments d'une liste L, ainsi que la *méthode* `random.shuffle` : `random.shuffle(L)` mélange les éléments de L (L est modifiée, ne crée pas une nouvelle liste).

Et maintenant l'itération :

EXERCICE 5

1. Écrire une fonction `RecalculeCluster(X, G, k)` qui recalcule les clusters comme décrit dans les étapes 2 et 3 de l'algorithme ci-dessus. Cette fonction devra renvoyer de nouvelles listes C et G, ainsi qu'un booléen qui indique si les clusters (ou les isobarycentres) ont été ou non modifiés.
2. Enfin, écrire la fonction `KMEAN` qui réalise le partitionnement.
3. Vous aurez peut-être constaté, lors des essais, que le programme s'arrête parfois avec une erreur, due au fait qu'un cluster est vide (dans ce cas, la recherche de l'isobarycentre échoue).
   Modifier votre procédure `RecalculeCluster(X, G, k)` pour prendre ce cas en considération (voir https://stackoverflow.com/questions/11075272/k-means-empty-cluster pour plus de détail).

Pour aller plus loin :

Si les programmes précédents sont terminés et valides, des développements supplémentaires intéressants sont possibles :

- **1.** Une fois le partitionnement fait, il faut affecter à chaque cluster une étiquette (ce sera l'étiquette majoritaire des éléments du cluster) puis à tester le bon fonctionnement de votre programme, en faisant afficher la matrice de confusion : il s'agit ici d'une matrice 6×6, où à l'indice (i, j) on trouve le nombre d'objets de catégorie i que le programme a classé en catégorie j.
- **2.** Un suprême raffinement : malheureusement, très souvent, l'algorithme converge vers un minimum local, qui dépend fortement des isobarycentres choisis au départ. Pour obtenir un programme valide, il faut donc l'appliquer un certain nombre de fois, et choisir parmi les solutions obtenues celle qui minimise la quantité la quantité $\sum_{j=0}^{k-1} \sum_{M \in C_j} d(M, G_j)^2$.
- **3.** Enfin, vous avez peut-être été déçu des résultats (personnellement mon taux de réussite maximum a été de 87,5%). Cela peut venir du fait que les données ne sont pas tou à fait du même ordre de grandeur, et par suite, certaines prennent plus d'importance que d'autres. On peut remédier à cela en les normalisant, c'est-à-dire en appliquant à chaque donnée (hauteur, poids, etc...) une fonction affine qui transforme l'intervalle $[\min(donnees)\,;\max(donnees)]$ en l'intervalle $[0\,;1]$. J'ai obtenu ainsi un taux de réussite légèrement supérieur à 90%.