Intelligence Artificielle - Application en médecine

I Présentation

Voir l'énoncé. Pas de question posée.

II Analyse des données

Q1 La structure de la table MEDICAL (`id` dédié et clé étrangère `idpatient`) montre que pour 1 patient donné, il peut y avoir plusieurs lignes dans MEDICAL. Comme on peut se passer de jointure, on prévoit un DISTINCT dans cette requête :
SELECT DISTINCT `idpatient` FROM MEDICAL
WHERE `etat` = "hernie discale"

Q2 **SELECT `nom, prenom` FROM PATIENT**
JOIN MEDICAL ON PATIENT.`id` = MEDICAL.`idpatient`
WHERE `etat` = "spondylolisthésis"

Q3 **SELECT `etat`, COUNT(`DISTINCT idpatient`) FROM MEDICAL**
GROUP BY `etat`

Q4 D'un point de vue général, `numpy` est un module qui fournit des outils appropriés aux **opérations matricielles**.
Par ailleurs, dans `numpy`, **on peut allouer la place strictement nécessaire au stockage en mémoire des tableaux**, typiquement en accompagnant la création des `arrays` de `numpy` par un `data type`. Ainsi, pour la table `etat`, on utilisera `dtype = numpy.uint8` afin de stocker exactement 8 bits et pour la table `data`, on utilisera `dtype = numpy.float32` afin de stocker les réels en multi-précision sur 32 bits selon la norme IEEE 754.

Q5 Quantité de mémoire nécessaire pour stocker `data` :
On donne directement le résultat en octets (i.e. 8 bits).
Les réels sont stockés sur 32 bits, soit 4 octets, en norme IEE 754, donc ils sont stockés en format simple précision. [1]
Chaque ligne nécessite $4 \times n$ octets. Il y a $N =$ lignes. Il faut donc $4nN$ octets.
Avec $n = 6$ et $N = 100\ 000$, il faut $4 \times 6 \times 100\ 000 = 2,4$ Mo.

Quantité de mémoire nécessaire pour stocker `etat` :
Chaque ligne nécessite 1 octet. Il y a N lignes. Il faut donc N octets.
Avec $N = 100\ 000$, il faut $0,1$ Mo.

Il faut donc au total $\boxed{2,5 \text{ Mo}}$.

Q6

```
15 def separationParGroupe(data, etat):
16     '''data est un array à N lignes et n colonnes et etat est un array à
       N lignes et 1 colonne'''
17     G = [[], [], []] # data des patients dont l'état est resp. 0, 1, 2
18     for i in range(len(data)):
19         G[etat[i]].append(data[i])
20     return G
```

1. en format simple précision, il y a 1 bit de signe, 8 bits d'exposant et 23 bits de mantisse

Q7 — `plt.figure()` affiche un nouvel écran dans lequel vont apparaître les figures.

— `plt.subplot(a, b, k)` considère l'écran divisé en $a \times b$ « petites » cases.
Il affiche la figure sur la k^e case (en comptant du haut à gauche vers la droite sur la première ligne, ceci ligne après ligne, en finissant en bas à droite).

— Les lignes et les colonnes sont respectivement numérotées de 0 à $n-1$ ([2]ce que montre le code en page 5). Or, $k = n \times (k//n) + (k\%n)$ où $k//n = i$ est le quotient et $k\%n = j+1$ est le reste. Ainsi, $k = n \times i + (j+1)$.
Ainsi, on obtient `ARGS1` :

```
47 n, n, n*i+j+1
```

— `plt.scatter` doit permettre de tracer l'attribut i de `data` en fonction de j.
On récupère la i^e colonne de `groupes[k]` et la j^e colonne de `groupes[k]`.
Ainsi, on obtient `ARGS2` :

```
50 groupes[k][:,i], groupes[k][:,j], marker = mark[k]
```

— `plt.hist` nécessite uniquement les données de la colonne i de `data` (le choix du nombre de barres est automatique). On ne peut malheureusement pas partir de `groupes` (comme suggéré par l'énoncé en utilisant deux fois la variable `datax`, car `groupes` est constitué de 3 tableaux distincts), donc on utilise `data`.
Ainsi, on obtient `ARGS3` :

```
53 data[:,i]
```

— Enfin, on trace un nuage de points uniquement quand i est différent de j.
Ainsi, on obtient `TEST` :

```
56 i != j
```

Q8 — Les diagrammes de la diagonale donnent la répartition d'un attribut donné parmi la population des patients de `data`.

— Les diagrammes hors diagonale montrent la corrélation entre deux attributs.

III Apprentissage et prédiction

III.1 Méthode KNN

Q9 L'énoncé suggère une transformation affine et suppose que $min(X) \neq max(X)$:

$$x_{normj} = \frac{x_j - min(X)}{max(X) - min(X)}$$

Q10 Nous proposons l'algorithme suivant dont la complexité est linéaire en la longueur de `X` :

```
63 def min_max(X):
64     m, M = X[0], X[0]
65     for i in range(len(X)):
66         if X[i] < m:
67             m = X[i]
68         elif X[i] > M:
69             M = X[i]
70     return m, M
```

2. en contradiction avec la légende de la FIGURE 3...

Q11

```
73 def d(u,v):
74     '''renvoie la distance entre les vecteurs u et v, supposés de même
       taille'''
75     n = len(u)
76     s = 0
77     for k in range(n):
78         s += (v[k] - u[k])**2
79     return s**0.5
80
81 def distance(z, data):
82     N = len(data)
83     D = []
84     for i in range(N):
85         D.append(d(z, data[i]))
86     return D
```

Q12 Le tri présenté est **le tri fusion (ou « merge sort »)**.

— Son **intérêt**[3] par rapport au tri par insertion réside dans sa **complexité dans le meilleur et dans le pire des cas**. En effet, celle-ci est en $\mathcal{O}(n\ log\ n)$, alors qu'elle est en $\mathcal{O}(n^2)$ pour le tri par insertion.

— Son **inconvénient** principal réside dans le fait que le tri fusion n'est **pas en place**, alors que le tri par insertion peut être réalisé en place.
Il y a cependant un inconvénient supplémentaire dans la version proposée du tri fusion. En effet, **la fusion effectuée par `fct` est récursive**. Or, si k est le nombre d'appels à `fct`, on a avec $p = \lceil log\ N \rceil$, $k > \sum_{i=1}^{p-2} 2^i = 2^{p-1} - 1 \geq \frac{N}{2} - 1$. Avec $N = 100\ 000$, **on dépasse largement le nombre maximal d'appels autorisé de base par Python (à savoir 1 000)**. Il serait préférable ici d'utiliser une version itérative de la fusion.

Q13

```
89 return T2 # ligne 1
90 return T1 # ligne 2
91 return [T2[0]] + fct(T1, T2[1:]) # ligne 3
```

Q14 Description de l'algorithme KNN proposé :

— **`La partie 1` fournit une liste de patients dont les données médicales sont les K données les plus proches de `z`.**

— **`La partie 2` détermine le nombre de patients dans chacun des états (3 dans l'exemple) parmi les K patients dont les données sont les plus proches de `z`.**

— **`La partie 3` détermine l'état le plus fréquent parmi les K patients dont les données sont les plus proches de `z`.**

Description des variables locales :

— `T` est la liste des éléments de la forme `[d, i]` où d est la distance de `z` aux données du patient d'indice i dans `data`.

— **`dist`** est la liste des distance de `z` aux données de chaque patient.

3. Il y a d'autres avantages au tri par insertion, peu pertinents ici :

— il est efficace lorsque les données sont déjà partiellement triées (on peut se rapprocher d'une complexité en $\mathcal{O}(n)$, du tri dans le meilleur des cas),

— il est structurellement bien adapté lorsque les données arrivent au fil de l'eau.

— `select` est la liste du nombre de patients par état, où l'on ne considère ici que les K patients dont les données sont les plus proches de `z`.

— `ind` est l'état le plus fréquent parmi les K patients dont les données sont les plus proches de `z`.

Q15 **La « matrice de confusion »** donne sur sa diagonale le nombre de patients du jeu de données de test (`datatest` associé à `etattest`) dont l'état est celui prédit par le modèle KNN (basé sur `data`).

Sur la première ligne,

— 23 patients du jeu de test ont un état 0, alors que le modèle prédisait un état 0,

— 4 patients du jeu de test ont un état 0, alors que le modèle prédisait un état 1,

— 7 patients du jeu de test ont un état 0, alors que le modèle prédisait un état 2.

Sur la première colonne,

— 23 patients du jeu de test ont un état 0, alors que le modèle prédisait un état 0,

— 7 patients du jeu de test ont un état 1, alors que le modèle prédisait un état 0,

— 5 patients du jeu de test ont un état 2, alors que le modèle prédisait un état 0.

Q16 La courbe obtenu donne le pourcentage de réussite (total de la diagonale/total de la matrice) en fonction de K.
Quand le nombre de voisins croît de 1 à 8, la taux de réussite croît pour atteindre un maximum vers 74%. Ensuite, il varie légèrement pour atteindre à 11, à nouveau le maximum. Enfin, le pourcentage de réussite décroît pour de plus grandes valeurs de K.
Donnons quelques explications :

— Pour de petites valeurs de K, on n'exploite pas suffisamment les données et on est sensible à des erreurs de mesure.

— Pour de grandes valeurs de K, on exploite trop de données, y compris des données loin du patient à tester, et on dégrade les résultats.

Notons par ailleurs que, plus K est grand, plus les performances seront dégradées par le grand nombre de valeurs à traiter. Ainsi, parmi les 2 valeurs a priori optimales $K = 8$ et $K = 11$, il est préférable de choisir $K = 8$.
Finalement, selon la courbe présentée, **le choix de $K = 8$ est optimal pour l'algorithme KNN**.

III.2 Méthode de classification naïve bayésienne

Q17

```
102 def moyenne(x):
103     ''' on suppose ici x non vide'''
104     s = 0
105     for i in range(len(x)):
106         s += x[i]
107     return s/len(x)
108 
109 def variance(x):
110     ''' on suppose ici x non vide'''
111     moy = moyenne(x)
112     s = 0
113     for i in range(len(x)):
114         s += (x[i] - moy)**2
115     return s/len(x)
```

Q18 Il ne faut pas oublier de prendre la racine carrée de la variance afin de produire un écart-type.

```
118 from math import sqrt
119
120 def synthese(data, etat):
121     groupes = separationParGroupe(data, etat)
122     groupes = np.array(groupes, dtype = object) # ajouté car le code
    donné ne fonctionne pas
123     for i in range(len(groupes)):
124         groupes[i] = np.array(groupes[i]) # groupe[i] devient un array
    numpy
125
126     n = np.shape(groupes[0])[1]
127     nb = np.shape(groupes)[0]
128
129     Probas = [[0 for j in range(n)] for i in range(nb)]
130
131     for i in range(nb):
132         for j in range(n):
133             Probas[i][j] = [moyenne(groupes[i][:,j]), sqrt(variance(
    groupes[i][:,j]))]
134
135     return Probas
```

Q19 La moyenne et la variance étant connues, on applique la formule donnée en page 10.

```
138 from math import sqrt, exp, pi
139
140 def gaussienne(a, moy, v):
141     return (1/sqrt(2*pi*v))*exp(-(a-moy)**2/(2*v))
```

Q20 Il faut comprendre que le tableau produit par `synthese(data,etat)` donne pour chaque couple $(X_i = x_i, y_j)$ (avec $i \in [\![1,6]\!]$ et $y_0 = 0, y_1 = 1, y_2 = 2$), les valeurs de la moyenne μ_{x_i,y_j} et de la variance $\sigma^2_{x_i,y_j}$ à utiliser pour le calcul de $P(X_i = z_i|Y = y_j)$.
On fait ainsi le calcul pour chacun de ces couples à y_j constant (donc par ligne) et on multiplie les résultats entre eux.

```
144 def probabiliteGroupe(z,data,etat):
145     M = synthese(data, etat)
146     nb, n = len(M), len(z)
147     Probas = [0 for _ in range(nb)]
148
149     for etat in range(nb):
150         proba = 1
151         for k in range(n):
152             moy, v = M[etat][k]
153             proba *= gaussienne(z[k], moy, v)
154         Probas[etat]  = proba
155
156     return Probas
```

Q21 On récupère l'état dont la probabilité est la plus élevée en utilisant la question précédente.

```
159 def prediction(z,data,etat):
160     Probas = probabiliteGroupe(z,data,etat)
161     etat = 0
162     M = Probas[etat]
163     for i in range(len(Probas)):
164         if Probas[i] > M:
165             M = Probas[i]
166             etat = i
167     return etat
```

Q22 Les valeurs des probabilités obtenues sont faibles : le logarithme permet d'utiliser **des nombres plus grands**, ce qui **améliore la répartition des données** et **évite d'avoir trop de nombres proches de la précision machine**.
Notons que cela nécessitera de remplacer le produit en somme dans la fonction `probabiliteGroupe`.

Q23
- Le taux de réussite pour la méthode KNN avec $K = 8$ est de 74% (somme de la diagonale de la matrice, divisée par 100).
- Le taux de réussite pour la méthode bayésienne est de l'ordre de 68% (exactement $\frac{2}{3}$).

Les résultat sont **en faveur de la méthode KNN sur l'exemple traité**.
Il manque trop d'informations pour discuter de façon générale de la pertinence de l'une ou l'autre méthode.