Informatique - CCP PSI 2019

Partie II – Analyse des données

Q1 - Écrire une requête SQL permettant d'extraire les identifiants des patients ayant une « hernie discale ».

```
SELECT idpatient FROM MEDICAL WHERE etat= « herniediscale »
```

Q2- Écrire une requête SQL permettant d'extraire les noms et prénoms des patients atteints de « spondylolisthésis».

```
SELECT nom, prenom FROM PATIENT JOIN MEDICAL ON
PATIENT.id=MEDICAL.idpatient WHERE MEDICAL.etat =
« spondylolisthésis »
```

Q3- Écrire une requête SQL permettant d'extraire chaque état et le nombre de patients pour chaque état.

```
SELECT etat,COUNT(*) FROM PATIENT JOIN MEDICAL ON
PATIENT.id=MEDICAL.idpatient GROUP BY etat
```

Ou plus simplement:

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

Q4 – Citer un intérêt d'utiliser la bibliothèque de calcul numérique Numpy quand les tableaux sont de grande taille.

Les temps de calcul sont beaucoup plus courts.

Q5 – Déterminer la quantité de mémoire totale en Mo nécessaire pour stocker le tableau et le vecteur des données si N=100 000. On supposera que les données sont représentées en suivant la norme usuelle IEEE 754.

Tableau `data` : $N \times 6 \times \frac{32}{8} = 2\,400\,000 \text{ o} = 2{,}4 \text{ Mo}$

Vecteur `etat` : $N \times 1 = 100\,000 \text{ o} = 0{,}1 \text{ Mo}$

Soit au total $2{,}5 \text{ Mo}$

Q6 – Écrire une fonction `separationParGroupe(data,etat)` qui sépare le tableau `data` en 3 sous-tableaux en fonction des valeurs du vecteur `etat` correspondantes. Cette fonction doit renvoyer une liste de taille 3 de sous-tableaux.

```
def separationParGroupe(data,etat) :
    return [[data[i] for i in range(len(data)) if etat[i]==j]
for j in range(3)]
```

Autre possibililité :

```
def separationParGroupe(data,etat):
    res = [[], [], []]
    for i in range(len(data)):
```

```
            res[etat[i]].append(data[i])
      return res
```

```
fig = plt.figure()
mark = ['o','x','*']
label_attributs=['incidence_bassin','orientation_bassin','angl
e_lordose','pente_sacrum','rayon_bassin','glissement_spon']
groupes=separationParGroupe(data,etat)
for i in range(len(groupes)):
     groupes[i]=array(groupes[i])
n=len(data[0])
for i in range(n):
     for j in range(n):
          ax1=plt.subplot(ARGS1)
          plt.xlabel(label_attributs[j])
          if TEST:
               plt.ylabel(label_attributs[i]
               for k in range(len(groupes)):
                    ax1.scatter(ARGS2)
          else:
               plt.ylabel('nombre de patients')
               ax1.hist(ARGS3)
```

remarques : Il y a une inversion dans le sujet entre xlabel et ylabel.
De plus, la définition du label des ordonnées ne devrait pas être dans la boucle d'indice k.

Q7 – Définir les arguments ARGS1, ARGS2, ARGS3 ainsi que la condition TEST définis dans le script précédent permettant d'obtenir la figure 2.

```
ARGS1= n,n,n*i+j+1
TEST = (j!=i)
ARGS2 = groupes[k][:,i],groupes[k][:,j],marker=mark[k]
ARGS3= data[:,j]
```

Q8 - Préciser l'utilité des diagrammes de la diagonale ainsi que celle des diagrammes hors diagonale.

Les diagrammes sur la diagonale donnent la répartition des patients en fonction de l'attribut j.
Les autres diagrammes donnent un attribut en fonction d'un autre, ils permettent de voir s'il y a un lien de corrélation entre 2 attributs. De plus ils permettent de visualiser les répartitions de ces attributs en fonction de l'état de l'individu.

Partie III – Apprentissage et prédiction

III-1- Méthode KNN

Q9 – Proposer une expression de x_{normj} un élément du vecteur X_{norm} en fonction de l'élément x_j du vecteur X correspondant et de min(X) et max(X).

On suppose une transformation linéaire soit :

$x_{normj} = a\,x_j + b$ avec $0 = a\min(X) + b$ et $1 = a\max(X) + b$

D'où $a = \frac{1}{\max(X)-\min(X)}$ et $b = -\frac{\min(X)}{\max(X)-\min(X)}$

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

Q10 – Écrire une fonction `min_max(X)` qui retourne les valeurs du minimum et du maximum d'un vecteur `X` passé en argument. La fonction devra être de complexité linéaire.

```
def min_max(X) :
     mini,maxi = X[0],X[0]
     for e in X :
          if e < mini :
               mini = e
          if e > maxi :
               maxi = e
     return mini,maxi
```

Q11 – Écrire une fonction `distance(z,data)` qui parcourt les N lignes du tableau `data` et calcule les distances euclidiennes entre le n-uplet `z` et chaque n-uplet `x` du tableau de données connues (`x` représente une ligne du tableau). La fonction doit renvoyer une liste de taille N contenant les distances entre chaque n-uplet `x` et le n-uplet `z`.

1. Si les données sont déjà normalisées :

```
def distance(z,data) :
     distances =[]
     for i in range(len(data)) :
          d=0
          for j in range(len(z)) :
               d +=((z[j]-data[i][j]))**2
          distances.append(sqrt(d))
     return distances
```

2. Si les données ne sont pas normalisées :

```
def coeff_normalise(data):
     L=[]
     for i in range(len(data[0])):
          colonne=[data[j][i] for j in range(len(data))] # si data
     n'est pas un tableau numpy ; sinon colonne = data[:,i]
          mini,maxi = min_max(colonne)
          a = 1/(maxi-mini)
          b = -mini/(maxi-mini)
          L.append([a,b])
     return L

def distance(z,data) :
     coef = coeff_normalise(data)
     distances = []
```

```
    for i in range(len(data)) :
        d=0
        for j in range(len(z)) :
            a = coef[j][0]
            d +=((z[j]-data[i][j])*a)**2 #distance normalisée
        distances.append(sqrt(d))
    return distances
```

```
def KNN(data,etat,z,K,nb):
    #partie 1
    T=[]
    dist=distance(z,data)
    for i in range(len(dist)):
        T.append([dist[i],i])
    #tri(T)
    T = tri(T)

    #partie 2
    select = [0]*nb
    for i in range(K):
        select[etat[T[i][1]]]+=1
    #partie 3
    ind = 0
    res = select[0]
    for k in range(1,nb):
        if select[k]>res:
            res=select[k]
            ind=k
    return ind
```

Q12 – Expliquer ce que font globalement les parties 1, 2 et 3 de l'algorithme. Préciser ce que représentent les variables locales `T, dist, select, ind`.

partie1 : construit une liste de listes contenant la distance euclidienne entre les attributs du patient à classer et les attributs de chaque patient déjà classé, ainsi que l'indice du patient classé. Cette liste est triée par ordre des distances croissantes.
`T` : liste de listes des distances et indices
`dist` : liste des distances entre les attributs du patient à classer et ceux des patients déjà classés

partie 2 : comptabilise, parmi les K premiers patients, le nombre de patients pour chaque état.
`select` : liste de **nb** éléments, chaque élément i contient le nombre de patients (parmi les K premiers) possédant l'état i.

partie 3 : recherche l'état correspondant au groupe le plus nombreux
`ind` : numéro de l'état possédant le groupe le plus nombreux

Q13 – Indiquer l'information apportée par la diagonale de la matrice. Exploiter les valeurs de la première ligne de cette matrice en expliquant les informations que l'on peut en tirer. Faire de même avec la première colonne. En déduire à quoi sert cette matrice.

Les informations de la diagonale indiquent le nombre de prédictions correctes pour chaque état (de 0 à 2).
Pour la première ligne : 4 patient 0 (normaux) ont été détectés 1 (hernie discale) et 7 ont été détectés 2 (spondylolisthésis) soit 11 faux diagnostiques parmi les 34 normaux.
Pour la première colonne : 7 patients possédant une hernie discale (état 1) et 5 souffrant de spondylolisthésis (état 2) n'ont pas été détectés.
Cette matrice permet d'observer l'efficacité du diagnostic et de cibler les erreurs.

Q14 – Commenter la courbe obtenue et critiquer l'efficacité de l'algorithme.

Le nombre de voisins K influe peu sur l'efficacité qui reste inférieure à 75%.

III-2- Méthode de classification naïve bayésienne

Q15 – Écrire deux fonctions de complexité linéaire `moyenne(x)` et `variance(x)` permettant de renvoyer la moyenne et la variance d'un vecteur x de dimension quelconque.

```
def moyenne(x) :
    moy = 0
    for e in x :
        moy += e
    return moy/len(x)

def variance(x) :
    var=0
    m = moyenne(x)
    for e in x :
        var += (e-m)**2
    return var/len(x)
```

Q16 – Proposer une fonction `synthese(data,etat)` qui renvoie une liste composée de doublets [moyenne,variance] pour chaque attribut de la matrice `data` en les regroupant selon les valeurs du vecteur `etat`.

```
def synthese(data,etat) :
  res = []
  groupes = separationParGroupe(data,etat)
  for k in range(len(groupes)): #indice etat
     groupes[k] = array(groupes[k])
     l=[]
     for j in range(len(data[0])): #indice attribut
       x = groupes[k][:,j]
       l.append([moyenne(x),variance(x)])
       res.append(l)
  return res
```

Q17 – Écrire une fonction `gaussienne(a,moy,v)` qui calcule la probabilité selon une loi gaussienne de moyenne moy et de variance v pour un élément a de $\mathbb{R}$.

```
def gaussienne(a,moy,v) :
    return 1/sqrt(2*pi*v)*exp(-(a-moy)**2/(2*v))
```

Q18 - Déduire de la description de l'algorithme de classification naïve bayésienne une fonction `probabiliteGroupe(z,data,etat)` prenant en argument le vecteur `z` qui contient le n-uplet de la donnée z, le tableau de données connues `data` et le vecteur d'appartenance à un groupe de chacune de ces données `etat`. Cette fonction renvoie la probabilité d'appartenance à chacun des $nb = 3$ groupes sous la forme d'une liste de trois valeurs. On utilisera la fonction `synthese(data,etat)` et la fonction `gaussiennes(a,moy,v)`.

```
def probabiliteGroupe(z,data,etat) :
    liste = []
    synt = synthese(data,etat)
    groupes = separationParGroupe(data,etat)
    for k in range(3) :
        proba = 1
        for j in range(len(data[0])) :
            a = z[j]
            moy = synt[k][j][0]
            v = synt[k][j][1]
            proba *= gaussienne(a,moy,v)
        PY = len(groupes[k])/len(data)          # P(Y=yj)
        proba *= PY
        liste.append(proba)
    return liste
```

Q19 – Écrire une fonction `prediction`, dont vous préciserez les arguments, qui renvoie le numéro du groupe auquel appartient un élément z.

```
def prediction(z,data,etat) :
    prob = probabiliteGroupe(z,data,etat)
    i,maxi = 0,prob[0]
    for k in range(1,3) :
        if prob[k]>maxi :
            i,maxi = k,prob[k]
    return i
```

Q20 – Proposer une explication qui justifie cette utilisation du logarithme.

Les probabilités sont très petites ; précision du codage
Cela permet de remplacer les multiplications de petites valeurs (entre 0 et 1) par des additions de valeurs plus grandes.

Q21 - Calculer le pourcentage de réussite de la méthode KNN, à partir de la matrice de confusion pour $K = 8$, ainsi que celui de la méthode naïve bayésienne à partir de la matrice ci-dessus. Discuter de la pertinence de chacune des deux méthodes sur l'exemple traité.

Pourcentage de réussite méthode KNN :

(23+11+40)/(23+4+7+7+11+1+5+2+40) = 74/100 = 74%

Pourcentage de réussite méthode bayésienne :

(23+10+49)/(23+9+8+9+10+1+10+1+49)=82/120=68% il y a plus de 100 valeurs… ce qui semble étrange

Si on raisonne sur les 2 matrices de confusion, la méthode KNN est plus efficace. Ce n'est pas ce qu'on observe sous python