TP 8

SUJET CCINP - PSI - 2019

Q1 – On part du principe qu'un patient n'apparaît qu'une seule fois.

```
SELECT idpatient FROM MEDICAL WHERE etat = "hernie discale"
```

Si un patient pouvait apparaître plusieurs fois, il faudrait ajouter `DISTINCT` :

```
SELECT DISTINCT idpatient FROM MEDICAL WHERE etat = "hernie discale"
```

Q2 –

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

Q3 –

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

Q4 – Pour une liste, la taille peut changer donc Python alloue plus d'espace que nécessaire pour stocker la liste.
Dans le cas d'un array, comme la taille est fixe, Python alloue exactement la taille nécessaire au stockage.
La place mémoire est donc économisée.
Notons que les temps de calculs sont également plus intéressants avec les array.

Q5 – La variable `data` contient N × n flottants codés sur 32 bits soit (pour N = 100000 et $n = 6$) : $\frac{100000\times6\times32}{8} = 2400000$ octets.
La variable `etat` contient N entiers codés sur 8 bits soit : $\frac{100000\times8}{8} = 100000$ octets.
Il y a donc besoin de 2500000 octets soit 2,5 Mo.

Q6 –

```
def separationParGRoupe(data, etat):
    L = [[], [], []]
    for i in range(len(data)):
        L[etat[i]].append(data[i, :])
    return L
```

Q7 – Il y a une interversion des `xlabel` et des `ylabel`.
À la ligne 13, il s'agit de choisir le graphique à la ligne i et colonne j parmi $n \times n$ graphiques (en numérotant à partir de 1 en haut à gauche) :

```
ax1 = plt.subplot( n, n, i*n+j+1)
```

À la ligne 15, on vérifie si l'on est hors de la diagonale :

```
if i != j:
```

À la ligne 18, on trace l'attribut i en fonction de j pour l'état k, il s'agit donc de récupérer les colonnes i et j du tableau `groupe[k]` :

```
ax1.scatter( groupes[k][:,i], groupes[k][:,j], marker = mark[k])
```

Enfin, à la ligne 21, on crée un histogramme à partir de la colonne j de `data` :

```
ax1.hist(data[:,j]
```

Q8 – Les diagrammes sur la diagonale donnent la répartition d'un attribut dans la population.
Les autres indiquent la corrélation entre deux attributs.

Q9 – On fait une transformation linéaire :

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

Q10 –

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

Q11 –

```
def distance(z, data):
    N, n = shape(data)
    D = zeros(N) # on pourrait aussi utiliser une liste
    for i in range(N):
        for k in range(n):
            D[i] += (data[i,k] - z[k])**2
        D[i] = sqrt(D[i])  # ou np.sqrt... mais la racine n'est pas essentielle pour la
            suite
    return D
```

Q12 – C'est le tri fusion et la fonction `fct` réalise la fusion.
La complexité de ce tri est quasi-linéaire (*i.e.* $\mathrm{O}(n\ln(n))$) alors que le tri par insertion est quadratique (*i.e.* $\mathrm{O}(n^2)$) donc le tri fusion est plus rapide.
Cependant ce tri n'est pas en place et demande de stocker des copies de parties du tableau.

Q13 –

```
def fct(T1 , T2):
    if T1 == []:
        return T2      # ligne 1
    if T2 == []:
        return T1      # ligne 2
    if T1[0][0] < T2[0][0]:
        return [T1[0]] + fct( T1[1,:], T2)
    else:
        return [T2[0]] + fct( T1, T2[1,:])    # ligne 3
```

Q14 – La variable `dist` est la liste des distances.
La partie 1 crée la liste `T` des couples de la forme `[dist(i), i]`, pour chaque patient, et effectue le tri de cette liste en fonction de la distance.
Dans la partie 2, on compte pour chaque état combien de patients sont dans cet état dans le groupe des K plus proches patients.
Après cela, `select[k]` contient le nombre de patients dans l'état `k` parmi les `K` plus proches de `z`.
Pour la partie 3, on recherche l'indice `ind` le plus grand dans `select` *i.e.* on recherche l'état avec le plus grand nombre de patients parmi les `K` plus proches de `z`.

Q15 – Sur la diagonale on a le nombre d'états bien déterminés par l'algorithme.

La première ligne indique :

- 23 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 0 , ont été détectés dans l'état 0 par l'algorithme ;
- 4 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 0 , ont été détectés dans l'état 1 par l'algorithme ;
- 7 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 0 , ont été détectés dans l'état 2 par l'algorithme.

La première colonne indique :

- 23 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 0 , ont été détectés dans l'état 0 par l'algorithme ;
- 7 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 1 , ont été détectés dans l'état 0 par l'algorithme ;
- 5 patients (de la base `dataset`) dont on sait qu'ils sont dans l'état 2 , ont été détectés dans l'état 0 par l'algorithme.

Cette matrice permet de quantifier la qualité de la détection et de déterminer le nombre d'erreurs.

Q16 – Le taux de succès semble croître jusqu'à K = 8, stagner jusqu'à K = 13 puis décroître.

Si l'on prend peu de voisins alors on exploite peu les données et on risque d'être sensible aux erreurs de mesures.

Si l'on prend K grand alors on exploite des points loin de celui considéré.

Notons également que les grandes valeurs de K ralentissent l'algorithme.

La partie sur l'algorithme des *k* plus proches voisins se termine là (hormis une comparaison à la fin). La suite du sujet propose une autre méthode moins dans l'esprit du programme.

Q17 –

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

def moyenne_bis(x):
    return sum(x)/len(x)

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

def variance_bis(x):
    ''' entrée : x de type array '''
    m = moyenne(x)
    return sum((x-m)**2)/len(x)
```

Q18 –

```
def synthese(data, etat):
    groupes = separationParGroupe(data, etat)

    nb = len(groupes)
    for k in range( nb):
        groupes[k] = array(groupes[k])

    NG, n = len(groupes[0])  # nb d'individus dans le groupe, nb d'attributs

    listeMoyVar = [ [ [0,0] for j in range(n)] for k in range(nb)]

    for k in range(nb) :
        for j in range(n) :
            mu, sig = moyenne(groupes[k][:,j]), sqrt(variance(groupes[k][:,j]))
            listMoyVar[k][j] = [mu, sig]

    return listMoyVar
```

Q19 –
```
def gaussienne(a, moy, v):
    return exp( -(a-mu)**2 / (2*v) ) / sqrt(2*pi*v)
```

Q20 –
```
def probabiliteGroupe(z, data, etat):
    lsyn = synthese(data, etat) # liste de liste de couple
    nb = len(lsyn) # nbr d'état (3 sur l'exemple)
    n = len(z) # nbr d'attributs
    lproba = [ 0 for i in range(nb) ]
    for etat in range(nb):
        proba = 1
        for i in range(n):
            moy, sig = lsyn[etat][i]
            proba *= gaussienne(z[i], moy , sig**2)
    lproba[etat] = proba
    return lproba
```

Q21 –
```
def prediction(z, data, etat):
    lproba = probabiliteGroupe(z, data, etat)
    etat = 0
    for i in range(len(lproba)):
        if lproba[i] > lproba[etat]:
            etat = i
    return etat
```

Q22 – L'utilisation d'un logarithme peut être pertinente car on utilise de petites valeurs (entre 0 et 1) et cela permet de remplacer les produits (de petites valeurs) par des sommes.

Q23 – Il y a plus de 100 valeurs ! Il s'agit peut-être d'un 29 en bas à droite.

La méthode KNN donne 74% de succès.

Cette méthode donne 68% (62% si c'est un 29 au lieu du 49).

La matrice de confusion semble indiquer que KNN est plus rapide.