CORRIGÉ DM INFO n°3 (CCINP PSI 2019)

Partie II. Analyse de données

❏ **Q1.** Une sélection toute simple :

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

❏ **Q2.** Une sélection avec jointure des deux tables :

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

ou bien :

```
SELECT p.nom, p.prenom FROM PATIENT AS p, MEDICAL AS m
   WHERE m.idpatient = p.id AND etat = "spondylolisthesis"
```

❏ **Q3.** On utilise la fonction d'agrégation `COUNT` :

```
SELECT etat, COUNT(idpatient) AS "Nombre de patients" FROM MEDICAL
   GROUP BY etat
```

On pouvait supposer qu'un patient n'apparaît pas deux fois avec la même pathologie, sinon il aurait fallu écrire : `COUNT(DISTINCT idpatient)`.

❏ **Q4.** Lorsque la taille d'un ensemble de données est fixe, il peut être intéressant d'utiliser `numpy` car alors, Python alloue la taille exacte des données en mémoire.
Alors que l'utilisation d'une liste, gérée dynamiquement, utilise plus de place mémoire que nécessaire.

De plus, l'utilisation de `numpy` permet d'accélérer les opérations de calcul matriciel et de pouvoir faire des opérations directement sur les vecteurs, ce qui n'est pas possible avec les listes (si besoin, je ne sais pas encore si ce sera le cas ici !).

❏ **Q5.** Le tableau `data` comporte $N \times n$ variables codées chacune sur 32 bits=4 octets, et le tableau `etat` contient N données codées chacune sur 1 octet .

Soit au total : $N \times n \times 4 + N = 2\,500\,000$ octets, c'est-à-dire 2,5 Mo.

❏ **Q6.** Il y a plusieurs possibilités d'écrire le programme demandé, selon qu'on utilise plus ou moins les facilités de Python. Une solution possible est la suivante.

```
def separationParGroupe(data, etat):
    nb = len(etat) # vaut 3 ici
    N = len(data) # nombre de lignes du tableau cad nombre de patients
    L = [ [] for _ in range(nb) ] # liste de listes vides
    for i in range(N):
        L[etat[i]].append(data[i])
    return L
```

❏ **Q7.** On notera l'erreur ligne 8 du programme fourni : il aurait fallu écrire `np.array` et non `array` (bien sûr, cela dépend comment a été importé `numpy`, mais il y a bien écrit `np.zeros` dans l'exemple de la question 15).

De plus, compte tenu des exemples figurant en haut de la page 5, il semble que dans le programme fourni, les labels sur les axes Ox et Oy ont été inversés !

- Ligne 13 : il faut choisir le graphe de coordonnées (i, j), avec $(i, j) \in [\![0\,;n-1]\!]^2$. Le numéro de la figure correspondante est donc $n * i + j + 1$, puisque la numérotation commence à 1. Il faut donc écrire :

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

- Ligne 15 : il s'agit juste de tester si on est ou pas sur la diagonale :

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

- Ligne 18 : l'énoncé dit que le graphique situé à l'emplacement (i,j) représente l'attribut i en fonction de l'attribut j. Mais dans le programme fourni, l'attribut j se trouve sur l'axe Oy et l'attribut i sur l'axe Ox (cf. lignes 14 et 17). Je pense donc que les `plt.xlabel` et `plt.ylabel` ont été inversés...

  Ces attributs se trouvent dans les colonnes de numéros i et j du tableau `groupes[k]`, donc on écrira :

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

- Ligne 21 :Là encore, sur l'exemple figurant en haut de la page 5, le label « Nombre de patients » est celui de l'axe Oy, il aurait donc fallu écrire `plt.ylabel` et non `plt.xlabel`.

  Les données dont on veut tracer l'histogramme sont celles de la colonne i du tableau `data` :

```
ax1.hist( data[:, i] )
```

❏ **Q8.** Les diagrammes sur la diagonale permettent de visualiser la répartition d'un attribut dans la population étudiée.

Ceux hors diagonale permettent de visualiser une éventuelle corrélation entre deux attributs.

Partie III. Apprentissage et prédiction

III.1 - Méthode KNN

❏ **Q9.** Il s'agit d'une transformation affine : $\min(X) \to 0$ et $\max(X) \to 1$ d'où la formule :

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

❏ **Q10.** Un grand classique...

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

❏ **Q11.**

```
def distance(z, data):
    N, n = data.shape # ou bien : N=len(data) et  n=len(data[0])
    dist = []
    for i in range(N):
        dist.append( np.sqrt( sum( [ (z[j] - data[i,j])**2 for j in range(n) ] ) ) )
        # en utilisant mieux numpy, il suffit d'écrire: sum( (z - data[i,:])**2 ) )
    return dist
```

❏ **Q12.** Il s'agit du tri fusion, cf cours pour le détail des avantages (rapidité) et inconvénients (encombrement mémoire).

Rem : bien sûr, il ne s'agissait pas ici de commenter la façon particulièrement laide dont est ici implémentée la méthode !

❏ **Q13.** Il s'agit ici de la fusion des deux listes T1 et T2, implémentée (*horreur !*) de façon récursive.

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

Cette écriture récursive est évidemment à proscrire, car si les deux listes sont de taille $m = n//2$, il pourra y avoir n appels récursifs imbriqués, donc très rapidement un dépassement de la taille de la pile allouée par Python. On préfèrera bien sûr l'algorithme itératif vu en cours.

On peut aussi remarquer, dans la fonction `tri(T)`, la méthode *affreuse* qui est utilisée pour découper une liste en deux ; pourquoi ne pas avoir utilisé `T[:m]` et `T[m:]` plutôt que ces deux boucles ?

❏ **Q14.**
- La première partie de la fonction `KNN` construit la liste T des `[dist[i], i]` pour tous les patients dans le tableau `data` (`dist[i]` représentant la distance entre les données du patient z et celui de numéro i), puis trie la liste selon les distances croissantes.

  On notera une erreur dans le programme (encore..) : le tri fusion ne s'effectuant pas en place, mais renvoyant la liste triée, il aurait fallu écrire `T = tri(T)` et non seulement `tri(T)`.
- La deuxième partie compte le nombre de patients dans chaque état, parmi les K plus proches voisins retenus. Le résultat est dans la liste `select`, `select[k]` étant le nombre de patients dans l'état k.
- Enfin, la troisième partie détermine l'état où il y a le plus grand nombre de patients. Cet état est dans la variable `ind`.

❏ **Q15.** Le terme d'indice (i, j) de la matrice (avec $i, j \in [\![0\,;2]\!]$) donne le nombre de patients dans l'état i et qui ont été détectés dans l'état j.

En particulier, les termes diagonaux donnent le nombre de patients qui ont été correctement détectés.

Par exemple, la 1ère ligne correspond au nombre de patients dans l'état 0 ; 23 d'entre eux ont été correctement détectés, 4 d'entre eux ont été indiqués en état 1 et 7 d'entre eux ont été détectés en état 2.

On peut vérifier que le total des nombres dans la matrice est bien égal à 100 (nombre total de patients).

Cette matrice permet donc de quantifier la précision de l'algorithme.

❏ **Q16.** D'après la figure, le pourcentage de réussite est d'abord croissant puis décroissant. Cela s'explique ; en effet, si l'on prend K trop petit, on ne considère pas suffisamment de données donc le résultat est imprécis, mais si l'on prend K trop grand, on considère des patients trop éloignés du patient à tester donc on introduit des cas peu probables.

On peut cependant noter que le pourcentage varie peu, entre 70 et 75% quelle que soit la valeur de K, cela signifie donc que cet algorithme est peu probant.

III.2 - Méthode naïve bayesienne

❏ **Q17.** Là encore, plusieurs réponses sont possibles en fonction des possibilités Python utilisées.

- Méthode simpliste :

```
def moyenne(x):
    s = 0
    n = len(x)
    for i in range(n):
        s += x[i]
    return s / n
```

- Méthode un peu moins simpliste :

```
def moyenne(x):
    s = 0
    for elt in x:
        s += elt
    return s / len(x)
```

- Encore mieux :

```
def moyenne(x):
    return sum(x) / len(x)
```

- Et on pourrait carrément utiliser la fonction `np.mean` de `numpy` !
- Pour le calcul de la variance, on peut écrire par exemple :

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

- ou encore, en utilisant le calcul vectoriel permis par `numpy`, et la formule de Koenig-Huyghens :

```
def variance(x):
    return moyenne(x**2) - moyenne(x)**2
```

Notons que, pour la suite, il eût été plus pratique d'écrire une fonction qui renvoyât simultanément moyenne et variance, cela évitait de calculer 2 fois la moyenne.

❏ **Q18.** Encore une erreur d'énoncé ! En effet, il est indiqué qu'il faut mettre dans la liste `data` les listes `[moyenne, variance]`, mais dans l'exemple donné en bas de la page on voit apparaître les couples $[\mu_{x_i,y_j}, \sigma_{x_i,y_j}]$ avec σ l'écart-type !! Je chosirais donc de stocker la *variance* car cela est plus cohérent avec le reste de l'énoncé.

Pour le début de la fonction, on s'inspire de ce qui a été fait avant :

```
def synthese(data, etat):
    groupes = separationParGroupe(data, etat)
    nb = len(groupes) # vaut 3 ici
    L = [ [] for _ in range(nb) ] # liste de listes vides pour le résultat
    # conversion en array numpy pour pouvoir extraire facilement les colonnes:
    for i in range(nb):
        groupes[i] = np.array( groupes[i] )

    n = len( data[0] )
    for i in range(nb):
        for j in range(n):
            x = groupes[i][:, j]
            L[i].append( [moyenne(x),  variance (x)] )
    return L
```

❏ **Q19.** On recopie la formule de l'énoncé (toujours en supposant que `numpy` a été importé *proprement*, c'est-à-dire par : `import numpy as np`.

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

❏ **Q20.**

```
def probabiliteGroupe(z, data, etat):
    synth = synthese(data, etat)
    nb = len(synth) # vaut 3
    L = [ 1 for _ in range(nb) ] # pour les probas résultats
    N, n = data.shape
    # on commence par calculer les probabilités des évènements (Y = yj)
    # et pour cela on calcule le nombre de personnes dans chaque état
    probas = [0]*nb
    for i in range(N):
        probas[etat[i]] += 1
    for j in range(nb):
        probas[j] /= N
    # on calcule enfin P(Y=yj) fois le produit des P(Xi=zi | Y=yj)
    for j in range(nb):
        for i in range(n):
            mu, vari = synth[j][i]
            L[j] *= gaussienne( z[i], mu, vari ) * probas[j]
    return L
```

❏ **Q21.** Il suffit de déterminer l'indice de la probabilité maximale.

```
def prediction( z, data, etat):
    L = probabiliteGroupe(z, data, etat)
    ind = 0
    for i in range( 1, len(L) ):
        if L[i] > L[ind]:
            ind = i
    return ind
```

❏ **Q22.** Je vois deux explications possibles, qui sont d'ailleurs liées.

- le logarithme permet de remplacer les produits par des sommes
- et les quantités étant très petites (de l'ordre de 10^{-13} selon l'exemple), on diminuera ainsi les erreurs de calcul.

❏ **Q23.** J'avoue être un peu perplexe... En effet, la matrice de confusion de l'énoncé concerne 120 patients (total des valeurs de la matrice) alors que l'énoncé parle de 100 personnes...

Dans la matrice de la page 8, le taux de réussite est de $\dfrac{23+11+40}{100} = 74\%$ alors que dans la nouvelle matrice il est de $\dfrac{23+10+49}{120} \approx 68\%$.

Tout cela n'est guère concluant, surtout si cela conduit à un diagnostic capable de bouleverser la vie du patient ! On peut penser que c'est à cause du nombre de données, trop faible.