Devoir d'informatique n° 3 concours blanc : correction

Q1.
```sql
SELECT idpatient
FROM MEDICAL
WHERE etat = 'hernie discale';
```

Q2.
```sql
SELECT nom, prenom
FROM PATIENT AS p
JOIN MEDICAL AS m ON p.id = m.idpatient
WHERE m.etat = 'spondylolisthésis';
```

Q3.
```sql
SELECT etat, COUNT(idpatient)
FROM MEDICAL
GROUP BY etat;
```

Q4. Le tableau `data` contient $N = 100\,000$ lignes et $n = 6$ colonnes de réels codés sur 32 bits chacun, ce qui nécessite $100\,000 \times 6 \times 32 = 192 \times 10^5$ bits. Le vecteur `etat` contient $N = 100\,000$ entiers codés sur 8 bits chacun, ce qui nécessite $100\,000 \times 8 = 8 \times 10^5$ bits. Il faut donc $192 \times 10^5 + 8 \times 10^5 = 2 \times 10^7$ bits pour stocker les deux. Or $\dfrac{2 \times 10^7}{8} = 2{,}5 \times 10^6$ d'où le $\boxed{\text{besoin de 2,5 Mo}}$ pour stocker le tableau et le vecteur des données.

Q5. On parcourt le vecteur `etat` et suivant la valeur (0, 1 ou 2) de la ligne i, on met les données correspondantes du patient (*i.e.* la i-ème ligne du tableau `data`) dans l'un des trois sous-tableaux (on se sert de la syntaxe `tableau[i,:]` donnée en annexe pour extraite une ligne d'un tableau) :

```
1  def separationParGroupe(data: array, etat: array) -> [list]:
2      N, n = data.shape    # ou juste N = len(data)
3      normal = []
4      hernie = []
5      spondy = []
6      for i in range(N):
7          if etat[i] == 0:
8              normal.append(data[i,:])
9          elif etat[i] == 1:
10             hernie.append(data[i,:])
11         else:
12             spondy.append(data[i,:])
13     return [normal, hernie, spondy]
```

Q6. Pour `ARGS1` à la ligne 13, il s'agit de donner le numéro du graphique situé à la i-ème ligne et j-ème colonne sachant qu'ils sont numérotés de 1 à $n \times n$ de gauche à droite et de haut en bas :

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

Pour `ARGS2` à la ligne 18, il s'agit de tracer l'attribut j (qui est une colonne de `groupes[k]`, syntaxe en annexe pour la récupérer facilement) en fonction de l'attribut i pour l'état k dont le symbole se trouve dans la liste `mark` :

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

Pour `TEST` à la ligne 15, il s'agit de tester si on se trouve en dehors de la diagonale :

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

Pour `ARGS3` à la ligne 21, il s'agit de tracer l'histogramme de l'attribut $i = j$:

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

Q7. Les histogrammes de la diagonale permettent de visualiser la répartition des valeurs de chaque attribut au sein de l'ensemble des patients.
Les graphiques en dehors de la diagonale permettent quant à eux de visualiser de quelle manière un attribut dépend d'un autre, s'il y a corrélation ou non entre les deux.

Q8. Une façon de ramener l'intervalle $[\min(X)\,;\max(X)]$ vers l'intervalle $[0\,;1]$ est de soustraire $\min(X)$ puis de diviser par la longueur de l'intervalle. Autrement dit

$$x_{norm,\,j} = \frac{x_j - \min(X)}{\max(X) - \min(X)}.$$

Q9. C'est une question très classique.

```
def min_max(X: [float]) -> (float, float):
    assert len(X) > 0  # Pour éviter erreur ensuite si liste vide.
    mini, maxi = X[0], X[0]
    for x in X:
        if x < mini:
            mini = x
        elif x > maxi:
            maxi = x
    return mini, maxi
```

Q10. On calcule la distance entre le vecteur `z` et le vecteur `data[i]` pour chaque ligne i du tableau.

```
def distance(z: tuple, data: array) -> [float]:
    N, n = data.shape  # ou N = len(data) et n = len(z)
    distances = []
    for i in range(N):
        dist = 0
        for k in range(n):
            dist = dist + (data[i,k] - z[k])**2
        distances.append(dist**0.5)
    return distances
```

Q11. Après la ligne 8, `tmp1` contient la première moitié de la liste `T` (les éléments d'indice 0 à $\frac{\texttt{len(T)}}{2}$, ce dernier exclu). Après la ligne 11, `tmp2` contient la seconde moitié de la liste `T` (ceux dont l'indice est supérieur ou égal à $\frac{\texttt{len(T)}}{2}$).

Q12. À la ligne 19, on compare le premier élément de `T1[0]` au premier élément de `T2[0]`. Or ces premiers éléments sont la distance entre le n-uplet à classer et le n-uplet connu. Ainsi, cette comparaison consiste à trouver le n-uplet le plus proche du n-uplet connu parmi les premiers éléments de chacune des deux listes `T1` et `T2`.

Q13. Les deux premières lignes à compléter correspondent à des cas triviaux : si une des deux listes est vide, pour effectuer la fusion il suffit de renvoyer l'autre liste. Pour la troisième, il s'agit du cas symétrique à celui de la ligne 20 : on prend le premier élément de `T2` et on appelle `fct` avec la liste `T2` privée de son premier élément.

```
def fct(T1, T2):
    if T1 == []:
        return T2
    if T2 == []:
        return T1:
    if T1[0][0] < T2[0][0]:  # Le plus petit élément est le 1er de T1.
        return [T1[0]] + fct(T1[1:], T2)
    else:  # Le plus petit élément est le 1er de T2.
        return [T2[0]] + fct(T1, T2[1:])
```

Q14. Si une liste est de taille 0 ou 1, elle est déjà triée (lignes 2 et 3). Sinon, on coupe cette liste en deux parties égales (à un élément près si elle est de longueur impaire, lignes 5 à 11). On trie alors récursivement chacune de ces moitiés et on fusionne ces moitiés triées grâce à la fonction `fct` (ligne 12). En pratique, on va ainsi couper la liste de départ jusqu'à des morceaux de taille 1 qui sont alors trivialement des listes triées puis les fusionner deux à deux [1].
Remarque : il s'agit d'un tri fusion.

Q15. • La partie 1 consiste à créer une liste `T` de couples (distance, numéro du patient) triée selon la distance. En particulier, les K premiers éléments de `T` sont les K plus proches patients de z.

• La partie 2 consiste, pour chaque état, à compter le nombre de patients correspondants parmi les K plus proches patients de z.

• Enfin, la partie 3 consiste à chercher l'indice de l'état le plus présent parmi ceux dénombrés dans la partie 2.

• Concernant les variables :

– `T` est la liste des couples (distance, numéro du patient) ;
– `dist` est la liste des distances entre les patients et z ;
– `select` est une liste contenant le nombre de patients pour chaque état parmi les K plus proches de z ;
– `ind` est l'indice du maximum de la liste `select`, *i.e.* celui de l'état le plus commun parmi les K plus proches patients de z.

Q16. • Les éléments de la diagonale de la matrice représentent le nombre de bonnes prédictions par la méthode KNN pour chacun des trois états.

• Sur la première ligne, on peut lire que 23 (respectivement 4 et 7) patients dont on sait qu'ils sont dans l'état 0 ont été prédits dans l'état 0 (resp. 1 et 2).

• Sur la première colonne, on peut lire que 23 (resp. 7 et 5) patients dont on sait qu'ils sont dans l'état 0 (resp. 1 et 2) ont été prédits dans l'état 0.

• La matrice de confusion permet donc de déterminer, pour chacun des trois états, le nombre de réussites et d'échecs de prédiction, et ainsi de mesurer l'efficacité de la méthode KNN en détaillant les résultats pour chaque classe.

Q17. Le taux de bonne prédiction semble être maximum pour K compris entre 8 et 11 (proche de 75 % de réussite). Pour les valeurs plus basses ou plus hautes, ce taux est légèrement plus bas, il faut donc choisir la bonne valeur de K pour optimiser cette méthode.
De plus la précision reste relativement faible avec environ un quart d'erreurs.

Q18. Ce sont des fonctions du cours à maîtriser parfaitement, il s'agit essentiellement d'un calcul de somme.

```
1 def moyenne(x: list) -> float:
2     total = 0
3     for element in x:
4         total = total + element
5     return total / len(x)
```

Pour la variance, on fait attention de ne pas recalculer la moyenne à chaque fois afin de conserver une complexité linéaire :

```
1 def variance(x: list) -> float:
2     moy = moyenne(x)
3     total = 0
4     for element in x:
5         total = total + (element - moy)**2
6     return total / len(x)
```

1. Si besoin de l'illustrer, voir `http://demeslay.maths.free.fr/fichiers/info/old/Tris_cours_avancement.pdf` et `https://capytale2.ac-paris.fr/web/c/b766-268117`

Q19. L'énoncé est peu clair : il parle de variance mais écrit σ et non σ^2...

```
def synthese(data: array, etat: array) -> [[float, float]]:
    groupes = separationParGroupe(data, etat)
    # On convertit en tableau comme Q7 :
    nb_etats = len(groupes)
    for k in range(nb_etats):
        groupes[k] = array(groupes[k])
    N, n = data.shape
    tab_moy_var = zeros((nb, n)) # initialisation tableau sortie
    for k in range(nb_etats):
        for j in range(n):
            moy = moyenne(groupes[k][:,j])
            var = variance(groupes[k][:,j])
            tab_moy_var[k][j] = [moy, var]
    return tab_moy_var
```

Q20. On applique la formule donnée par l'énoncé (sachant que la variance vaut l'écart-type au carré).

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

Q21. Il s'agit de calculer le produit des probabilités pour chacun des trois groupes.

```
def probabiliteGroupe(z, data, etat) -> [float, float, float]:
    tab_moy_var = synthese(data, etat)
    n = len(z)  # nombre d'attributs
    liste_proba = []
    for k in range(3):
        proba = 1  # pour initialiser le produit
        for i in range(n):
            moy, var = tab_moy_var[k][i]
            proba = proba * gaussienne(z[i], moy, var)
        liste_proba.append(proba)
    return liste_proba
```

Q22. C'est au fond une question classique de recherche de l'indice du maximum.

```
def prediction(z, data, etat) -> int:
    liste_proba = probabiliteGroupe(z, data, etat)
    ind_maxi = 0
    for i in range(len(liste_proba)):
        if liste_proba[i] > liste_proba[ind_maxi]:
            ind_maxi = i
    return ind_maxi
```

Q23. D'une part, comme il s'agit de probabilités, ce sont des valeurs petites (les données de l'énoncé sont de l'ordre de 10^{-15} et donc proches de la précision machine), il est donc plus aisé de discuter les différences relatives en échelle logarithmique. D'autre part, cela permet de remplacer le calcul du produit par celui d'une somme, ce qui est souvent plus rapide.

Q24. Pour la méthode KNN dans le cas $K = 8$, il y avait $23 + 11 + 40 = 74$ bonnes prédictions sur 100 patients soit $\boxed{\text{un taux de réussite de } 74\ \%}$.
Pour la méthode naïve bayésienne, il y a $20 + 9 + 39 = 68$ bonnes prédictions sur 100 patients soit $\boxed{\text{un taux de réussite de } 68\ \%}$.
Ainsi, sur l'exemple traité, la méthode KNN pour $K = 8$ donne de meilleurs résultats que la méthode naïve bayésienne, elle semble donc plus pertinente.