CONCOURS BLANC « CCINP »

Correction

CCINP 2019 - PSI

Partie I - Présentation

Partie II - Analyse des données

Q1.
```
SELECT 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. Tous les éléments d'un tableau numpy ont le même type (par exemple `float64`), ils occupent donc tous la même taille en mémoire, python peut alors affecter exactement la bonne taille en mémoire et, si elle est contigu, cela ne coute rien d'accéder à n'importe quel élément.
Remarque : Dans le rapport du jury : « le module Numpy permet d'optimiser les opérations matricielles. Beaucoup de candidats se contentent de dire que cela permet uniquement d'avoir accès plus facilement à un élément ou que c'est optimisé en mémoire, voire que cela permettait de simplifier l'affichage dans la console. »
Je suis d'accord sur le fait que c'est adapté pour travailler avec des vecteurs et des matrices, mais cela optimise par rapport à quoi ? puisque que les algorithmes liés aux matrices pour les listes doivent être écrit (de manière naïf ou optimisé), de même le "se contente de" quand il est demandé "citer un intérêt".

Q5. On rappel qu'un octet (byte en anglais) contient 8 bits.
Dans `data` il y a Nn réels stockés comme des flottants sur 32 bits. Dans `etat` il y a N entiers sur 8 bits. On a donc besoin de $32Nn + 8N$ bits, on divise par 8 pour avoir en octet, puis par 1 000 000 pour avoir en Mo et on trouve (rappel $N = 100\,000$ et $n = 6$) 2.5 Mo

Q6. La fonction doit renvoyer `[L0,L1,L2]` où `L0` est la matrice extraite de `data` où on a gardé que les patients dans l'état 0, elle a le même nombre de colonne que `data` (ie n) et un certain nombre de lignes (la somme des nombres de lignes de `L0`, `L1` et `L2` doit valoir N).

```
def separationParGRoupe(data,etat):
    N,n = data.shape
    L=[]
    for k in [0,1,2]:
        L.append([data[i,:] for i in range(N) if etat[i] == k]
    return L

def separationParGRoupe(data,etat): #Si vous n'aimez pas les listes en compréhension
    #N,n = data.shape
    N,n = len(data), len(data[0]) # alternative
    L=[]
    for k in [0,1,2]:
        res=[]
        for i in range(N):
            if etat[i]==k:
                res.append(data[i,:])
        L.append(res)
    return L
```

Q7. On fera attention à la structure des données : `groupes` est une liste d'array, donc `groupes[k]` est un array
`ARGS1` : `n,n,i*n+j+1`
`ARGS2` : `groupes[k][:,i], groupes[k][:,j], marker = mark[k]`
`ARGS3` : `data[:,j]`
`TEST` : `i!=j`

Q8. Les diagrammes sur la diagonale indique la répartition d'un attribut, tandis que ceux hors de la diagonale montre la corrélation entre deux attributs.

Partie III - Apprentissage et prédiction

III.1 - Méthode KNN

Q9. On peut prendre $x_{normj} = \frac{x_j - \min(X)}{\max(X) - \min(X)}$.

Q10.
```
def min_max(X):
    tempo_min=X[0]
    tempo_max=X[0]
    for x in X:
        if x<tempo_min:
            tempo_min=x
        elif x>tempo_max:
            tempo_max=x
    return tempo_min,tempo_max
```

Q11. On utilise que pour le tableau numpy `data[i,:]` le carré correspond à mettre au carré tous ses éléments. Si on n'y pense pas, on n'oubliera pas d'initialiser la somme (2ème solution).

```
def distance(z, data ):
    N,n=data.shape
    dist=[] # On sait qu'il y aura N éléments mais il est demandé une liste
    for i in range(N):
        dist.append(sqrt(sum((data[i,:]-z)**2))
    return dist

def distance(z, data ):
    N,n=data.shape
    dist=[] # On sait qu'il y aura N éléments mais il est demandé une liste
    for i in range(N):
        res=0
        for j in range(n):
            res+=(data[i,j]-z[j])**2
        dist.append(sqrt(res))
    return dist
```

Q12. C'est le tri fusion, sa complexité ($O(n\ln(n))$) est meilleure que le tri par insertion ($O(n^2)$ dans les pires des cas), par contre il n'est pas en place (nécessite des copies du tableau à trier)

Q13. `ligne 1 : return T2`
`ligne 2 : return T1`
`ligne 3 : return [T2[0]] + fct(T1,T2[1,:])`

Q14.
- Dans la partie 1 on créé la liste `T` constituée de couples où on a le numéro du patient (en deuxième position) et sa distance à z. Pour le dire rapidement : on trie les patients en fonctions de la distance à z
- Dans la partie 2 on construit la liste `select` telle que `select[k]` contient le nombre de patient (parmi les K patients qui sont le plus proche de K) qui sont dans l'état K.
- Dans la partie 3 on cherche l'indice du plus grand élément de `select`, dit autrement on cherche l'état qui est le plus fréquent parmi les K plus proches voisins de z.

Q15. La diagonale de la matrice correspond au nombre de patient de la donnée test qui ont bien été classés par notre algorithme.
Sur la première ligne on a le résultat de notre algorithme sur les patients dans l'état 0 : 23 ont été bien classés, 4 ont été classés en état 1 et 7 en état 2.
• Sur la première colonne : 23 patients ont été bien classés, 7 ont été classés dans l'état 0 alors qu'ils sont dans l'état 1, 5 ont été mis dans l'état 0 alors qu'ils sont dans l'état 2.
Cette matrice est une mesure de la qualité de notre algorithme (s'il fonctionnait parfaitement la matrice serait diagonale) et on a le nombre d'erreurs (et où ils sont).

Q16. La qualité de l'algorithme commence par augmenter avec K jusque K=8, puis stagne jusque K=11, puis décroit. On voit l'importance du paramètre K, encore faut-il le déterminer. De plus il pourrait non seulement dépendre de nos données initiales (ce qui est normal) mais aussi de nos données de test (ce qui est plus gênant).

III.2 - Méthode de classification naïve bayésienne

Q17. Comme la complexité est imposée, il me semble naturel de la programmer complètement (attention si on dans la variance on recalcul la moyenne pour chaque passage dans la boucle on est en $O(n^2)$ et donc pas linéaire).

```
def moyenne(x):
    res=0
    n=len(x)
    for k in range(n):
        res+=x[k]
    return res/n
def variance(x):
    res=0
    n=len(x)
    m=moyenne(x)
    for k in range(n):
        res+=(x[k]-m)**2
    return res/n

# Alternative (sum est linéaire)
def moyenne(x):
    return sum(x)/len(x)
def variance(x):
    return sum((x-moyenne(x))**2)/len(x)
```

Q18.
```
def synthese(data,etat):
    N,n=data.shape

    # On commence par regrouper
    groupes = separationParGroupe (data , etat)
    nb = len(groupes)
    for k in range(nb):
        groupes[k] = array(groupes[k])

    res=[]
    for k in range(nb):
        res.append([])
        for j in range(n):
            mu=moyenne(groupes[k][:,j])
            sigma=sqrt(variance(groupes[k][:,j]))
            res[k].append([mu,sigma])
    return res
```

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

Q20.
```
def probabiliteGroupe(z,data,etat):
    lst_syn = synthese(data,etat)
    nb = len(lst_syn) # nbr d'état
    n = len(z) # nbr d'attributs
    lst_proba=[]
    for k in range(nb):
        proba=1
        for i in range(n):
            mu,sig = lst_syn[k][i]
            proba *= gaussienne(z[i],mu,sig**2)
        lst_proba.append(proba)
    return lst_proba
```

Q21. On recherche l'indice du plus grand élément d'une liste (ici s'il est réalisé plusieurs fois c'est l'état le plus petit qui est choisi)

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

Q22. Cela permet d'éviter d'avoir des nombres trop petit (au sens proche de 0) et mieux répartis. Ainsi on évite les overflow (plutôt les underflow)

Q23. Le pourcentage de réussite s'obtient à partir de la matrice de confusion en faisant le quotient de la somme des éléments diagonaux (ie les bonnes prédictions) par la somme de tous les coefficients de la matrice (ie le nombre de patients). On trouve 74% pour KNN et 68.3% pour la méthode naïve bayésienne. Ici KNN semble un peu plus efficace, mais on ne sait pas si c'est le hasard (données, données tests) ou non.