# CCPINP 2019

Pelletier Sylvain PSI, LMSC

## Partie II : Analyse de données

**Q1.** C'est une selection :

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

(on pourrait ajouter le mot clé DISTINCT si il y a plusieurs enregistrements pour un même patient dans la table MEDICAL).

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

**Q2.** Il faut faire une jointure avec la table PATIENT :

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

**Q3.** C'est une agrégation par état et l'application de la fonction d'agrégation COUNT pour chacun des états.
Si l'on considère qu'il n'y a qu'un enregistrement par patient dans MEDICAL :

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

sinon :

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

**Q4.** Dans le cas présent, c'est plutôt le fait que les tableaux prennent le minimum de place en mémoire qui est intéressant. Dans le cas d'une liste, comme la taille de la liste peut changer, 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.

**Q5.** On a $N \times n$ réels stockés sur 32 bits dans `data`. Dans `etat` on a $N$ entiers sur 8 bits. Au final, on a :

$$N \times n \times 32 + N \times 8$$

Avec $N = 100000$ et $n = 6$, cela donne : 20000000, ainsi 20Go.

**Q6.** On propose plusieurs méthodes :

```python
def separationParGRoupe(data, etat):
  """
  entrée: data =  2darray Nxn
               = données médicales
          etat = 1darray N
               = etat de chaque patient

  sortie: L0, L1, L2 = N array
               = données médicales pour chacun des états
  """
  N,n  = shape(data)

  # extraction d'une ligne ajouté par append
  L0 = []
  for i in range(N):
    if etat[i] == 0 :
      L0.append(data[i,:])

  # liste en compréhension
  L1 = [ data[i,:] for i in range(N) if etat[i] == 1]

  # idem  sans utiliser l'extraction de ligne
```

```
  L2 = []
  for i in range(N):
    if etat[i] == 2:
      Ltmp = []
      for k in range(n):
        Ltmp.append( data[i,k] )
      L2.append(Ltmp)

  return L0, L1, L2
```

Autre idée :

```
L = [ [] , [] , [] ] # les trois listes actuellement vides
for i in range(N):
  L[etat[i]].append( data[i,:] )
return L
```

**Q7.** Pour l'instruction l13, il s'agit de choisir le graphique ligne $i$ colonne $j$ dans un ensemble de $n \times n$ graphiques, sachant que les graphiques sont numérotés à partir de 1 et de haut gauche vers le bas droite. Cela donne donc ;

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

Pour l'instruction l18, il s'agit de tracer l'attribut $i$ en fonction de $j$ pour l'état $k$. Ces données se trouvent dans les colonnes $i$ et $j$ du tableau `groupe[k]`
Cela donne donc :

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

Pour le test l15, il s'agit de savoir si on n'est sur la diagonale :

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

Pour l'instruction l21, il s'agit de dessiner un histogramme de la colonne $j$ de `data` :

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

**Q8.** Les diagrammes sur la diagonale indique la répartition d'un attribut dans la population.
Les diagrammes hors de la diagonale montrent la corrélation entre deux attributs.

## Partie III : Apprentissage et prédiction

### III-1 Méthode KNN

**Q9.** On suppose que l'on fait une correction linéaire :

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

**Q10.** C'est très classique :

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

**Q11.** Il faut faire la distance euclidienne usuelle :

```
def distance(z, data):
  """
  entrée: z = 1d array de taille n
            = données sur un nouveau patient
          data = 2d array Nxn
            = données dans la base
```

```
  sortie: D = 1d array de taille N
8           = D[i] est la distance euclidienne entre les données du nouveau patient
                  et les données du patient i
10 """
  N,n = shape(data)
12 D = zeros(N)
  for i in range(N):
14   D[i] = sqrt( sum ( (data[i,:] - z)**2 ) )
```

Une autre solution (à la place de la dernière ligne) :

```
for i in range(N):
2  D[i] = 0 # inutile car D est initialisé avec des 0
  for k in range(n):
4    D[i] +=  (data[i,k] - z[k])**2
  D[i] = sqrt(D[i])
```

**Q12.** C'est un tri fusion, la fonction `fct` effectue la fusion. Ce tri est plus rapide (complexité en $O(n\ln(n))$) que le tri par insertion (complexité en $O(n)$), mais il n'est pas fait sur place : il est nécessaire de stocker créer des copies de parties du tableau (l6-8 et l9-12)

**Q13.** C'est l'algorithme de fusion qui est fait dans la fonction `fct` :

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

**Q14.** La partie 1 crée et trie une liste $T$ de couple de la forme `[dist(i), i]`, pour chaque patient $i$ enregistré dans `data`. Les couples sont triés en fonction de la distance. En regardant la deuxième composante de chaque couple, on peut alors connaître quels sont les patients les plus proches. En particulier, les $K$ premiers éléments de cette liste sont les $K$ patients les plus proches.
Dans la partie 2, on compte pour chaque état combien de patient sont dans cet état dans le groupe des $K$ plus proche patient. Après ces lignes, `select[k]` contiendra le nombre de patient dans l'état $k$ parmi les $K$ plus proches de $z$.
Pour les partie 3, on recherche l'indice le plus grand dans `select`, autrement dit, on recherche l'état avec le plus grand nombre de patient parmi les $K$ plus proches de $z$.
Ainsi :

- *dist* contient la liste des distances : `dist(i)` est la distance entre les données du patient $i$ et du patient $z$.
- $T$ contient une liste de couple de la forme `[dist(i), i]`, pour chaque patient $i$ enregistré dans `data`.
- `select` est une liste de la taille le nombre d'état : `select[k]` le nombre de patient dans l'état $k$ parmi les $K$ plus proches de $z$.
- `ind` est un indice dans `select` : c'est l'indice $k$ où `select[k]` est le plus grand, c'est donc l'état avec le plus grand nombre de patient parmi les $K$ plus proches de $z$.

**Q15.** Sur la diagonale on a le nombre d'état bien déterminé 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 ainsi de mesurer la qualité de la détection et de déterminer le nombre d'erreurs.

**Q16.** Le taux de réussite peut être considéré comme croissant jusqu'à $K = 8$, puis constant jusqu'à $K = 13$, puis décroissant.
L'idée est que :

- si on ne prend pas assez de voisins, alors on n'exploite pas suffisamment les données. Dans le cas extrême où on ne prend qu'un seul voisin, on est sensible aux erreurs de mesures.

- si on prend trop de voisins, alors on ajoute des fausses détections, puisqu'on prend en compte des patients « loin » du patient à tester.

Cet algorithme est donc critiquable car la valeur du paramètre $K$ est difficile à déterminer.

R Plus $K$ est grand plus l'algorithme est lent.

### III-3 Méthode de classification naïve bayésienne

**Q17.** C'est un classique de cours :

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

En supposant que l'on manipule des array, on peut écrire :

```
def variance(x) :
  n = len(x)
  m = moyenne(x)
  return sum( (x-m)**2) /n
```

**Q18.** Il faut découper en groupe puis appliquer les fonction précédentes sur chacun des groupes :

```
def synthese(data, etat) :
  """
  entrée: voir plus haut,
  sortie: listeMoyVar = liste de couples de taille nb x n
                        nb nbr de groupes
                        n nbr d'attributs
          = liste de la forme [moy, var]
            à la ligne k colonne j  donne la moyenne et variance de l'attribut j pour
  """

  # on récupère les groupes
  groupes = separationParGroupe(data, etat)
  # conversion comme Q7:
  nb = len(groupes)
  for k in range( nb):
    groupes[k] = array(groupes[k])

  # n est le nombre d'attribut (ne dépend pas du groupe)
  # NG  est le nombre d'individu dans le groupe
  NG, n = len(groupes[0])

  # initialisation de la sortie
  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.** Il suffit d'appliquer la formule (attention à la confusion variance / écart-type) :

```
def gaussienne(a, moy, v):
  """
  entrée: a = float = donnée
          moy = float = moyenne
          v = float = variance
  sortie: float = valeur de P(X=a) pour X qui suit une gaussienne
  """
  return exp( -(a-mu)**2 / (2*v) ) / sqrt(2*pi*v)
```

**Q20.** On peut faire :

```
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.** c'est une simple recherche de l'indice maximum :

```
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.** Plusieurs explications :

- On utilise des valeurs plutôt petite (entre 0 et 1).
- Cela permet de remplacer le produit par une somme.

**Q23.** Le pourcentage de réussite peut se calculer ainsi :

```
def tauxReussite(matConfusion) :
  """
  entrée: matConfusion = array nxn
                       = matrice de confusion
  sortie: taux   = float
                 = taux de réussite
  """
  n,m = shape(matConfusion)

  assert n == m, "matrice de confusion pas carrée"

  nbrPatient = 0
  for i in range(n):
    for j in range(n) :
      nbrPatient += matConfusion[i,j]

  nbrReussite = 0:
  for i in range(n):
    nbrReussite += matConfusion[i,i]

  taux = nbrReussite / nbrPatient
  return taux

M1 = array( [[23, 4, 7],
             [7, 11, 1],
             [5, 2, 40]] )
M2 = array( [[23, 9, 8],
             [9, 10, 1],
             [10, 1, 49]] )
print("taux de réussite KNN:",  tauxReussite(M1))
print("taux de réussite Bayésienne:",  tauxReussite(M2))
```