

Devoir Surveillé 2 - CORRECTION

Durée : 2 heures
Calculatrice interdite

I Distances sur les mots

Étant donné un mot, l'objectif de cet exercice est de savoir à quelle langue il appartient.

Pour cela, on dispose d'un ensemble de mots appartenant à chaque langue.

I. 1 Distance de Hamming

La distance de Hamming entre deux mots de même longueur est le nombre de positions où les deux mots sont différents. Par exemple, la distance de Hamming entre *arbre* et *arche* est 2 car il y a deux différences : à l'indice 2 ($b \neq c$) et à l'indice 3 ($r \neq h$). On rappelle qu'on peut accéder à la i -ème lettre d'une chaîne de caractères s avec $s[i]$ et qu'on peut connaître sa taille avec $\text{len}(s)$.

1. Écrire une fonction *hamming* (s, t) qui calcule la distance de Hamming entre deux mots de même longueur.

Par exemple, `hamming("arbre", "arche")` doit renvoyer 2.

```
def hamming(s, t):
    n = len(s)
    d = 0
    for i in range(n):
        if s[i] != t[i]:
            d += 1
    return d
```

I. 2 Plus proches voisins

On suppose avoir une liste X de mots dont les langues sont données par y ($y[i]$ est la langue du mot $X[i]$).

On commence par séparer X en deux ensembles X_{train} et X_{test} (et les langues correspondantes y_{train} et y_{test}).

2. Écrire une fonction *split*(L) renvoyant deux listes $L1$ et $L2$ séparant L en deux listes de même taille (à ± 1 près).

```
def split(L):
    n = len(L)
    L1, L2 = [], []
    for i in range(n):
```

```

    if i % 2 == 0:
        L1.append(L[i])
    else:
        L2.append(L[i])
return L1, L2

```

ou, plus rapidement, par la méthode dite du "slicing" :

```

def split(L):
    n = len(L)
    return L[:n//2], L[n//2:]

```

3. Expliquer quel est l'intérêt de séparer les données en deux ensembles avant d'utiliser un algorithme d'apprentissage.

L'algorithme est censé être utilisé sur de nouvelles données (que l'on ne connaît pas encore). Tester l'algorithme sur des données qu'il a déjà vues permet de savoir s'il est efficace sur de nouvelles données.

On suppose l'existence d'une fonction *voisins*(x, X_trains, k) permettant de trouver les indices des k plus proches voisins d'un mot x dans la liste de mots X_train (en utilisant, par exemple, la distance de Hamming).

Ainsi, si $L = \text{voisins}(x, X_trains, k)$ alors ;

- $L[0]$ est l'indice du mot le plus proche de x dans X_train
- $X_train[L[0]]$ est le mot correspondant (le mot le plus proche de x dans X_train).

4. Écrire une fonction *plus_frequent*(L) renvoyant l'élément le plus fréquent d'une liste L .

Par exemple, *plus_frequent* ([3, 4, 1, 1, 4, 3, 1]) doit renvoyer 1 .

On essaiera d'avoir la meilleure complexité possible.

```

def plus_frequent(L):
    d = {}
    for x in L:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
    kmin, vmin = 0, 0
    for k in d:
        if d[k] > vmin:
            kmin, vmin = k, d[k]
    return kmin

```

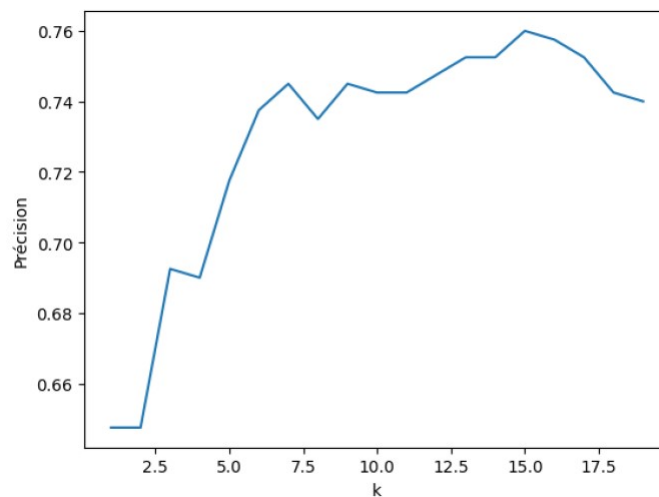
5. En déduire une fonction $knn(x,k)$ qui renvoie la langue majoritaire parmi les k mots les plus proches de x dans X_{train} .

```
def knn(x, k):  
    L = voisins(x, X_train, k)  
    return plus_frequent([y_train[i] for i in L])
```

6. Écrire une fonction $precision(k)$ qui renvoie la précision de l'algorithme knn pour une valeur de k donnée, en utilisant les données de test (X_{test}).

```
def precision(k):  
    n = len(X_test)  
    nb_correct = 0  
    for i in range(n):  
        if knn(X_test[i], k) == y_test[i]:  
            nb_correct += 1  
    return nb_correct / n
```

7. En calculant la précision pour différentes valeurs de k , on obtient la courbe suivante :



Donner (approximativement) l'erreur minimum que l'on peut obtenir avec l'algorithme des plus proches voisins.

Graphiquement, la précision maximum semble être 0.76 .
Donc l'erreur minimum est $1 - 0.76 = 0.24$.

8. On suppose avoir stocké les valeurs de précision dans un dictionnaire *precisions* dont les clés sont des valeurs de k et les valeurs les précisions correspondantes.
Écrire une fonction *meilleur_k(precisions)* qui renvoie la valeur de k qui donne la meilleure précision.

```
def meilleur_k(precisions):
    kmax = 1
    for k in precisions:
        if precisions[k] > precisions[kmax]:
            kmax = k
    return kmax
```

9. On applique l'algorithme des plus proches voisins avec deux langues : anglais (donné par l'entier 0 dans *y_train*) et français (donné par l'entier 1 dans *y_train*).

On obtient la matrice de confusion $\begin{pmatrix} 72 & 33 \\ 30 & 65 \end{pmatrix}$.

Dire à quoi correspond chacun des nombres de cette matrice. Quelle est la précision correspondante ?

- 72 est le nombre de fois où l'algorithme a prédit anglais et où la langue était bien anglais.
- 33 est le nombre de fois où l'algorithme a prédit anglais et où la langue était français.
- 30 est le nombre de fois où l'algorithme a prédit français et où la langue était anglais.
- 65 est le nombre de fois où l'algorithme a prédit français et où la langue était français.

La précision est donc $137/200 = 0.685$.

II Marché immobilier à Marseille

On stocke dans une matrice $M = (m_{i,j})$ les informations sur des appartements de Lyon, où chaque ligne correspond à un appartement et chaque colonne à une information : $m_{i,j}$ est la valeur de l'information j pour l'appartement i .

Prix (en millier d'euros)	Nombre de m2	Année de construction	Étage	DPE
300	50	2010	2	B
180	25	1990	1	C
⋮	⋮	⋮	⋮	

Par exemple, le tableau ci-dessus sera stocké dans une matrice de la forme suivante :

$$\begin{pmatrix} 300 & 50 & 2010 & \dots \\ 180 & 25 & 1990 & \dots \\ \vdots & \vdots & \vdots & \dots \end{pmatrix}$$

10. Le DPE (Diagnostic de Performance Énergétique) est une échelle de classement des logements en fonction de leur consommation d'énergie. Elle va de A à G, où A est le meilleur classement et G le pire.

Pourquoi est-ce important de convertir cette lettre en nombre, avant d'appliquer un algorithme d'apprentissage ?

Peut-on associer un nombre arbitraire à chaque lettre ?

Pour pouvoir calculer des moyennes, distances... on a besoin d'utiliser des réels. Il faut associer à chaque lettre un nombre, en respectant l'ordre des lettres.

Dans la suite, on supposera que les lettres ont été converties en nombres.

11. Écrire une fonction *moyenne*(M, j) renvoyant la moyenne des valeurs de la colonne j de la matrice M.

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

12. Écrire une fonction *ecart_type*(M, j) renvoyant l'écart-type des valeurs de la colonne j de la matrice M, c'est-à-dire $\sigma_j = \sqrt{\sum_i \frac{(m_{i,j} - \mu_j)^2}{n}}$ où μ_j est la moyenne de la colonne j de M.

On fera en sorte d'avoir une complexité linéaire en le nombre de lignes de M.

Il faut éviter de recalculer moyenne à chaque fois, en stockant le résultat dans une variable.

```
def ecart_type(M, j):
    n = len(M)
    mu = moyenne(M, j)
    s = 0
    for i in range(n):
        s += (M[i][j] - mu) ** 2
```

```

|   return (s / n) ** 0.5

```

13. Écrire une fonction **normalisation**(M) qui renvoie une nouvelle matrice $M' = (m'_{i,j})$ obtenue en normalisant toutes les colonnes de la matrice M , c'est-à-dire en soustrayant la moyenne et en divisant par l'écart-type : $m'_{i,j} = \frac{m_{i,j} - \mu_j}{\sigma_j}$.
Quelle est la complexité de cette fonction ?

```

| def normalisation(M):
|     n, p = len(M), len(M[0])
|     M2 = [[0 for i in range(p)] for i in range(n)]
|     for j in range(p): # O(np)
|         mu = moyenne(M, j) # O(n)
|         sigma = ecart_type(M, j) # O(n)
|         for i in range(n): # O(n)
|             M2[i][j] = (M[i][j] - mu) / sigma
|     return M2

```

On passe p fois dans le for j in range (p), qui exécute **moyenne** ($O(n)$), **ecart_type** ($O(n)$), et for i in range (n) ($O(n)$), ce qui fait au total une complexité $p \times (O(n) + O(n) + O(n)) = O(np)$.

14. Quelle est la moyenne et l'écart-type de chaque colonne de M' obtenue par normalisation ?

La moyenne de la colonne j de M est :

$$\frac{1}{n} \sum_i m'_{i,j} = \frac{1}{n} \sum_i \frac{m_{i,j} - \mu}{\sigma} = \frac{1}{\sigma} \sum_i \frac{m_{i,j}}{n} - \frac{\mu}{n\sigma} = \frac{\mu}{n\sigma} - \frac{\mu}{n\sigma} = 0.$$

L'écart-type de la colonne j de M est :

$$\sqrt{\frac{1}{n} \sum_i (m'_{i,j} - 0)^2} = \sqrt{\frac{1}{n} \sum_i \left(\frac{m_{i,j} - \mu}{\sigma}\right)^2} = 1.$$

Car $\sigma^2 = \frac{1}{n} \sum_i (m_{i,j} - \mu)^2$ par définition.

15. Pourquoi est-ce important de normaliser les données avant d'utiliser un algorithme d'apprentissage automatique ?

Sinon, un algorithme d'apprentissage automatique pourrait donner plus d'importance à certaines colonnes qu'à d'autres, car elles auraient des valeurs plus grandes.

Dans la suite, on suppose que M a été normalisée.

16. Écrire une fonction **distance**(u, v) qui renvoie la distance euclidienne entre les listes u et v (supposées de même taille). Cette dernière est

définie par : $\sqrt{\sum_j (u_j - v_j)^2}$.

```
def distance(u, v):
    s = 0
    for j in range(len(u)):
        s += (u[j] - v[j]) ** 2
    return s ** 0.5
```

17. Écrire une fonction *centre(X)* qui renvoie le centre de la matrice X, c'est-à-dire liste obtenue en calculant la moyenne de chaque colonne de la matrice X.

Par exemple, si X est la matrice :

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 5 & 0 \end{pmatrix}$$

alors centre(X) renverra : [2, 3.5, 1.5].

```
def centre(X):
    L = [0] * p
    for i in range(len(X)):
        for j in range(len(X[0])):
            L[j] += X[i][j]
    for j in range(p):
        L[j] /= n
    return L
```

Ou, en réutilisant *moyenne* :

```
def centre(X):
    L = []
    for j in range(len(X[0])):
        L.append(moyenne(X, j))
    return L
```

18. Écrire une fonction *calculer_centres* tel que, si classes est une liste de matrices, *calculer_centres(classes)* renvoie la liste des centres de chaque matrice de classes.

```
def calculer_centres(classes):
    L = []
    for X in classes:
        L.append(centre(X))
    return L
```

19. Écrire une fonction *calculer_classes* (*M*, *centres*) qui renvoie une liste *L* telle que *L*[*i*] soit la liste des lignes de *M* dont le centre le plus proche (au sens de distance) est *centres*[*i*].

```
def calculer_classes(M, centres):
    k = len(centres)
    L = [[] for i in range(k)]
    for i in range(len(M)):
        c = 0 # numero du centre le plus proche de M[i]
        d = distance(M[i], centres[0])
        for j in range(1, k):
            d2 = distance(M[i], centres[j])
            if d2 < d:
                c = j
                d = d2
        L[c].append(M[i])
    return L
```

20. Écrire une fonction *kmeans* (*M*, *k*) qui applique l'algorithme des *k*-moyennes à la matrice *M* et renvoie la liste des centres et des classes trouvés.

On initialisera les centres en prenant les *k* premières lignes de *M*.

```
def kmeans(M, k):
    centres = M[:k] # extrait les k premieres listes
    classes = calculer_classes(M, centres)
    while True:
        centres2 = calculer_centres(classes)
        classes2 = calculer_classes(M, centres2)
        if classes == classes2:
            return centres2, classes2
        centres = centres2
        classes = classes2
```