Informatique du Tronc Commun

TD n°4 Intelligence artificielle – *Corrigé*

Parcours possibles

♪ Si vous avez des difficultés en python : exercices n°1, n°2.

♪ ♪ Si vous vous sentez moyennement à l'aise, mais pas en difficulté : exercices n°1, n°2, n°3.

♪ ♪ ♪ Si vous êtes à l'aise : exercices n°3, n°4, n°5.

I Les briques élémentaires

Exercice n°1 Distances

R1. Écrire la fonction `dist(x:list,y:list)->float` qui renvoie la distance euclidienne au carré entre les deux vecteurs `x` et `y` de $\mathbb{R}^m$.

Solution:

```
def dist(x,y):
    d=0
    for i in range(len(x)):
        d=d+(x[i]-y[i])**2
    return d
```

R2. On connaît les coordonnées de Valence dans une liste `val=[..,..,..]`, on souhaite connaître la distance qui sépare Valence d'autres villes.

Nous avons une liste de listes `villes` de la forme `[ ['ville0',[..,..,..]],['ville1',[..,..,..]],...]`. Le premier élément de chaque sous-liste est le nom de la ville, le deuxième élément est la liste des coordonnées de la ville.

Remarque : c'est une structure quasi identique à celle du cours.

Écrire la fonction `liste_dist(val:list,villes:list[list])->list[list]` qui renvoie une liste de listes de deux éléments, le premier est le nom de la ville, le deuxième est la distance qui sépare cette ville à Valence.

Une structure possible pour la fonction, volontairement très détaillée pour vous guider (peut être écrite en moins de lignes, en en regroupant) :

```
def liste_dist(val,villes):
    L = .... #  initialisation de la liste des distances
    for i in range(.......):  # pour chaque ville
        d = ..................................... # calcul de la distance qui sépare Valence de la ville i, dont les coordonnées sont dans la liste ............
        N = ........................ # nom de la ville i
        L.append( [ ..... , ...... ] ) # on ajoute une liste de deux éléments à L
    return L
```

Solution:

```
def liste_dist(val,villes):
    L = []
```

```
    for i in range(len(villes)):  # pour chaque ville
        d = dist(val,villes[i][1]) # distance qui sépare la ville de
   Valence
        N = villes[i][0]
        L.append([N,d])
    return L
```

R3. On connaît les coordonnées de Valence dans une liste `val=[..,..,..]`, on souhaite connaître la distance qui sépare Valence d'autres villes.

Nous possédons deux listes, l'une avec les noms de ville `nom=['ville0,'ville1',...]` et l'autre avec les coordonnées de ces villes-là `coord=[ [..,..,..] , [..,..,..] , ... ]` dans le même ordre.

Écrire la fonction `liste_dist(val:list,nom:list,coord:list)->list[list]` qui renvoie une liste de listes de deux éléments, le premier le nom de la ville, le deuxième la distance qui sépare cette ville à Valence.

Solution:

```
def liste_dist(val,nom,coord):
    L = []
    for i in range(len(villes)):  # pour chaque ville
        d = dist(val,coord[i]) # distance qui sépare la ville de
   Valence
        N = nom[i]
        L.append([N,d])
    return L
```

R4. Écrire la fonction `dict_dist(val:list,nom:list,coord:list)->dict` qui renvoie un dictionnaire dont les clés sont le nom de la ville et les valeurs associées la distance qui sépare cette ville à Valence. *Les arguments de la fonction sont du même type que pour la question précédente.*

Solution:

```
def dict_dist(val,nom,coord):
    D = {}
    for i in range(len(villes)):  # pour chaque ville
        d = dist(val,coord[i]) # distance qui sépare la ville de
   Valence
        N = nom[i]
        D[N] = d # on crée la clé et on lui associe la valeur
    return D
```

Exercice n°2 Maximum ♥

R1. Écrire la fonction `rang_max(L:list)->int` qui prend en argument une liste `L` de flottants et renvoie le rang du maximum des valeurs de `L`.

Solution:

```
def rang_max(L):
    imax=0
    for i in range(1,len(L)):
        if L[i]>L[imax]: # nouveau maximum trouvé
            imax=i
    return imax
```

R2. Écrire la fonction `cle_dict_max(d:dict)->int` qui prend en argument un dictionnaire `d` dont les clés sont des chaines de caractères ou des entiers, et dont les valeurs associées sont des nombres, et qui renvoie la clé dont la valeur associée est maximale.

Solution:

```
def cle_dict_max(d):
    cmax = - float('inf')
    for c in d :
        if d[c]>d[cmax]: # nouveau maximum trouvé
            cmax = c
    return cmax
```

II Algorithme des k plus proches voisins

Exercice n°3 Les iris ♪♪

On utilise les données disponibles de la bibliothèque `Scikit-Learn` sur les iris. Les caractéristiques numériques (longueur du sépale, largeur du sépale, longueur du pétale, largeur du pétale) des iris de la base de données sont contenues dans le tableau `iris.data` à deux dimensions. `iris.data[i]` est un vecteur de $\mathbb{R}^4$ qui contient les quatre caractéristiques de l'iris `i`.
`iris.target[i]` est un entier (0, 1 ou 2) qui renvoie à la variété de l'iris `i`. Cet entier est le rang de la liste `iris.target_names` des variétés.

```
from sklearn.datasets import load_iris
iris=load_iris()
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
>>> iris.feature_names # liste des caractéristiques prises en compte
['sepal length(cm)','sepal width(cm)','petal length(cm)','petal width(cm)']
>>> iris.data # caractéristiques des différents iris
array([[5.1, 3.5, 1.4, 0.2] ,
       [4.9, 3. , 1.4, 0.2] , ... ])
>>> iris.target # donne la variété de l'iris (plus exactement son rang dans
   la liste iris.target_names) dans le même ordre que iris.data
array([0, 0 ,... , 2, 2 ])
```

L'objectif est de déterminer la variété d'un iris inconnu, connaissant les quatre caractéristiques (longueur du sépale, largeur du sépale, longueur du pétale, largeur du pétale.

R1. ♥ Écrire la fonction `dist(x:list,y:list)->float` qui renvoie la distance au carré entre les deux vecteurs `x` et `y` de $\mathbb{R}^4$.

Solution:

```
def dist(x,y):
    d=0
    for i in range(len(x)):
        d=d+(x[i]-y[i])**2
    return d
```

R2. ♥ Écrire la fonction `distances(C:array,V:array,X:array)->list` qui prend en arguments :

- `C` la liste de listes des caractéristiques des iris des données d'apprentissage (`C` est de la forme de `iris.data`),
- `V` la liste des variétés des données d'apprentissage (`V` est de la forme de `iris.target`)
- `X` la liste (de quatre éléments) des caractéristiques de l'iris dont on souhaite déterminer la variété,

et qui renvoie la liste de listes `D` de deux éléments :

- le premier élément de `D[i]` est la distance au carré entre `X` et `C[i]`,

— et le deuxième élément est la variété de l'iris `i`.

La liste renvoyée sera triée par ordre de distance croissante. Pour cela, on pourra utiliser `L.sort()` qui trie par ordre croissant `L` (par défaut, si `L` est une liste de listes, cela trie selon le premier élément de chaque sous liste).

Solution:

```
def distances(C,V,X):
    D=[]
    for i in range(len(C)):
        di=dist(C[i],X)
        D.append([di,i])
    return D
```

R3. ♥ Écrire la fonction `rang_max(L:list)->int` qui prend en argument une liste `L` de flottants et renvoie le rang du maximum des valeurs de `L`.

Solution:

```
def rang_max(L):
    imax=0
    for i in range(1,len(L)):
        if L[i]>L[imax]: # nouveau maximum trouvé
            imax=i
    return imax
```

R4. Écrire la fonction `kppvoisin(C:array,V:array,X:array,k)->int` qui prend en arguments `C` la liste des listes des caractéristiques des iris des données d'apprentissage (`C` est de la forme de `iris.data`), `V` la liste des variétés des données d'apprentissage (`V` est de la forme de `iris.target`), `X` la liste (de quatre éléments) des caractéristiques de l'iris dont on souhaite déterminer la variété, et `k` un entier qui est le nombre de plus proches voisins considérés, et qui renvoie la variété de l'iris inconnu (plus exactement son rang dans la liste `iris.target_names`) en utilisant l'algorithme des k plus proches voisins.

Solution:

```
def kppvoisin(C,V,X,k):
    D = distances(C,V,X) # liste des distances entre X et C
    D.sort() # on trie D par ordre de distance croissante
    n=[0]*3 # liste du nombre de voisins par variété
    for i in range(k): # parcours des k plus proches voisins
        v=D[i][1] # variété du voisin i
        n[v]+=1 # ajout de 1 à la variété
    #variété la plus représentée
    var_max=rang_max(n) # variété la plus représentée dans les k plus
  proches voisins
    return var_max
```

R5. Rappeler la définition de la matrice de confusion. Quelle est sa taille pour la situation qui nous intéresse ? Quelle information donnent les éléments sur la diagonale ? hors de la diagonale ?

Solution: En ligne sont présents la variété réelle des données de test, et en colonne la variété prédite par l'algorithme.

Ici, nous avons trois variétés, donc la matrice de confusion est une matrice 3×3.
Sur la diagonale se trouvent le nombre de prédictions conforment à la variété réelle. Hors de la diagonale est prédiction fausse.

R6. On obtient la matrice suivante : $\begin{pmatrix} 20 & 0 & 0 \\ 0 & 12 & 7 \\ 0 & 2 & 9 \end{pmatrix}$. Commenter.

Solution: L'algorithme détermine la bonne variété $20 + 12 + 9 = 41$ fois sur $20 + 12 + 9 + 7 + 2 = 50$, c'est-à-dire 82% des tests.
7 iris versicolor ont été prédits en virginica, et 2 virginicas ont étés prédits en versicolor. Ces deux variétés doivent être proches en terme de caractéristiques, et des confusions peuvent survenir.

Les données numériques sont d'ordres de grandeur différents selon les caractéristiques. Par conséquent, la longueur des sépales a une plus grande importance dans le classement que la largeur des pétales. Pour éviter cela, il faut normaliser les données, soit les ramener dans l'intervalle $[0, 1]$.
Pour cela, on envisage une normalisation linéaire telle que la largeur du sépale de l'iris i est modifiée comme suit :

$$d'_i = \frac{d_i - d_{\min}}{d_{\max} - d_{\min}}$$

, où $d_{\min}$ est la valeur minimale des largeurs des sépales de l'ensemble des iris, et $d_{\max}$ sa valeur maximale.
On adaptera la formule précédente pour toutes les caractéristiques.

R7. ♥ Écrire une fonction `min_max(T:list[list],i:int)->[float,float]` qui prend en argument un tableau `T` (liste de listes), et un entier `i` et qui renvoie une liste de deux éléments : la valeur minimale et la valeur maximale de la colonne de rang `i` de `T`.

Solution:

```
def min_max(T,i):
    m = T[0][i]
    M = T[0][i]
    for j in range(len(T)):
        if T[j][i] > M :
            M = T[j][i]
        elif T[j][i] < m :
            m = T[j][i]
   return [m,M]
```

R8. ♪ ♪ ♪ Proposer une fonction `normaliser` qui prend en argument la liste des données, et renvoie une nouvelle liste de même nature et dans le même ordre qui contient les données normalisées.

Solution:

```
def normaliser(L):
    Lnorm = [ [ 0 for i in range(len(L[0])) ] for j in range(len(L))]
    for i in range(len(L[0])) : # colonne
        m , M = min_max(L,i) # min, max de la colonne i
        for j in range(len(L)): # lignes de la colonne i
            Lnorm[j][i] = ( L[j][i] - m ) / (M-m)
    return Lnorm
```

Exercice n°4 Harry Potter ♪ ♪ ♪

À l'entrée à l'école de Poudlard, le Choixpeau magique répartit les élèves dans les différentes maisons (Gryffondor, Serdaigle, Serpentard et Poufsouffle) en fonction de leur courage, leur loyauté, leur sagesse et leur malice. Le Choixpeau magique se souvient de tous les anciens élèves depuis la création de Poudlard ainsi que de leurs caractéristiques.

Voici un tableau qui récapitule quelques élèves :

Nom	Courage	Loyauté	Sagesse	Malice	Maison
Hermione	8	6	6	6	Gryffondor
Drago	6	6	5	8	Serpentard
Cho	7	6	9	6	Serdaigle
Cédric	7	10	5	6	Poufsouffle
....					

Vous venez d'intégrer Poudlard, et le Choixpeau magique doit vous orienter dans la bonne maison.

La liste des élèves est supposée stockée dans un dictionnaire `eleves` où les clés sont le nom des élèves et les valeurs sont des tuples où la première coordonnée est une liste des valeurs des différentes caractéristiques de chaque élève et la deuxième coordonnée est la maison qui lui est attribuée. Ainsi :

```
eleves = { 'Hermione' : ( [8,6,6,6] , 'Gryffondor' ) ,
           'Drago'    : ( [6,6,5,8] , 'Serpentard' ),
         ....}
```

Afin de résoudre ce problème, nous allons appliquer la méthode des k plus proches voisins pour vous attribuer la maison majoritaire.

Nous définissons comme **distance entre deux élèves** la somme des valeurs absolues des différences de chaque caractéristique. Par exemple, la distance entre Cho et Cédric est de 8 car $|7-7|+|6-10|+|9-5|+|6-6| = 8$

R1. Écrire une fonction `dist(L1:list,L2:list)->float` qui prend en argument deux listes de caractéristiques de deux élèves différents, et renvoie la distance entre ces deux élèves.

Solution:

```
def dist(L1,L2):
    d = 0
    for i in range(4):
        d = d + abs(L1[i]-L2[i])
    return d
```

R2. Écrire une fonction `liste_dist(eleves:dict,vous:list)->list[list]` qui prend en argument le dictionnaire `eleves` de la liste des élèves qui renvoie une liste de listes où chaque élément `D[i]` est une liste de la forme `[d, nom_maison]` où `d` est la distance entre un élève du dictionnaire et vous.

Solution:

```
def liste_dist(eleves,vous):
    D = []
    for e in eleves :
        d = dist(vous,eleves[e][0])
        D.append([d,eleves[e][1]])
    return D
```

R3. Écrire une fonction `dic_nom_maison(v:list[list])->dict` qui prend en argument une liste `v` dont chaque élément `v[i]` est une liste `[d, nom_maison]` et qui renvoie un dictionnaire dont les clés sont les maisons et la valeur est le nombre d'élèves appartenant à cette maison dans la liste `v`.

Solution:

```
def dic_nom_maison(v):
    dico = {}
    for i in range(len(v)):
         = v[i][1]
        if n_mai in dico :
            dico[n_mai]+=1
        else:
            dico[n_mai]=1
    return dico
```

R4. Écrire une fonction `maximum(D:dict)->str` qui prend en argument un dictionnaire `D` et qui renvoie la clé dont la valeur est maximale.

Solution:

```
def maximum(D):
    M = - float('inf')
    for c in D:
        if D[c]>M:
            M = D[c]
            cle_M = c
    return cle_M
```

R5. Écrire une suite d'instructions (on n'attend pas une fonction) qui permet de savoir dans quel maison vous vous retrouverez dans le cas on nous considérons les 5 plus proches voisins.

Solution:

```
D = liste_dist(eleves,vous) # liste des distances
D.sort() # trie de la liste D par ordre croissant de distance
dic_maison = dic_nom_maison(D[:5]) # 5 premiers élèves les plus proches
    de vous
votre_maison = maximum(dic_maison)
```

III Algorithme des k moyennes

Exercice n°5 Compression d'image ♪ ♪ ♪

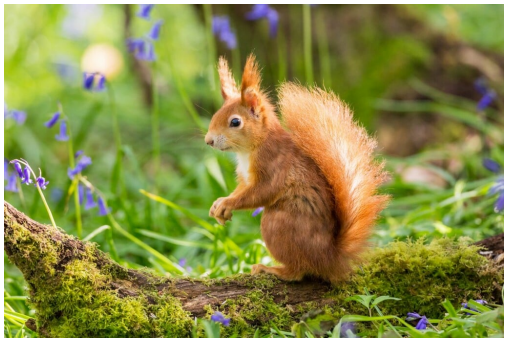

Image initiale : 267549 couleurs

Réduction avec 8 couleurs

Réduction avec 16 couleurs

On considère une photo de $m \times n$ pixels (m lignes, n colonnes).

Chaque pixel est représenté par un triplet (r, g, b) où r, g et b sont des entiers codés sur 8 bits (donc entre 0 et 255) représentants la quantité de rouge, vert et bleu du pixel. L'image est représentée en machine par un

tableau `numpy` à trois dimensions : le pixel de la ligne i et de la colonne j est représenté par une liste de trois entiers $[r, g, b]$ qui est l'élément de rang j de la liste de rang i.

R1. Écrire un script `python` qui calcule le nombre de couleurs différentes utilisées dans cette image.

On pourra créer un dictionnaire, dont les clés seront les 3-uplet caractérisant la couleur des pixels. Si la couleur n'a pas déjà été rencontrée (c'est-à-dire si le 3-uplet n'est pas dans le dictionnaire), il sera ajouté au dictionnaire (avec la valeur associée 1 par exemple).

Il restera à renvoyer la longueur du dictionnaire.

Solution: Un pixel est caractérisé par une liste de 3 entiers. Pour ne pas choisir plusieurs fois le même pixel, il faut vérifier qu'il n'a pas déjà été choisi. Pour cela, on stocke les pixels choisis dans un dictionnaire dont la clé est le pixel.

Cependant on ne peut pas placer une liste comme clé d'un dictionnaire. Une possibilité est de convertir la liste en chaîne de caractères avec `str()`.

```
def nb_couleurs(img):
    n,m = len(img) , len(img[0])
    d={} # dictionnaire qui stocke les couleurs (ie les triplets (r,g,b)) déjà rencontrées
     # clés de d : la couleur du pixel (r,g,b) ; valeurs : True (ou 1)
    for i in range(n):
        for j in range(m):
            if str(img[i][j]) not in d:  # accès en temps constant
                d[str(img[i][j])]=1 # on ajoute la couleur au dictionnaire
    return len(d) # le nombre de couleurs est le nombre d'éléments dans le dictionnaire
```

Les photos contiennent un très grand nombre de couleurs différentes. Notre objectif est de réduire ce nombre à seulement 16 couleurs. Pour ce faire, nous allons appliquer l'algorithme des k-moyennes pour regrouper les différents pixels en 16 classes, calculer la couleur moyenne de chacune de ces 16 classes, puis attribuer cette valeur moyenne à chacun des pixels de la classe correspondante.

La variable `img` est un tableau numpy à 3 dimensions : $m \times n \times 3$ et représente la photo que l'on veut compresser.

R2. Écrire une fonction `dist(p,q)` qui prend pour arguments deux pixels `p` et `q` (représentés par deux vecteurs dans $\mathbb{R}^3$) et renvoie la distance euclidienne entre ces deux vecteurs.

Solution:

```
def dist(p,q):
    d=0
    for i in range(3):
        d=d+(q[i]-p[i])**2
    return np.sqrt(d)
```

R3. Écrire une fonction `initialise(img,k)` qui prend pour arguments une image `img`, un entier `k` et renvoie un tableau `numpy` de k cases, chacune d'elles contenant un pixel tiré au hasard dans l'image.

On pourra utiliser la fonction `randint(a,b)` de la bibliothèque `random` qui renvoie aléatoirement un entier compris entre `a` inclus et `b` inclus.

Rq : on commencera par créer une liste de k éléments que l'on remplira comme indiqué, puis on finira par convertir la liste en tableau `numpy` avec `np.array(liste)`.

Solution: Un pixel est caractérisé par une liste de 3 entiers. Pour ne pas choisir plusieurs fois le même pixel, il faut vérifier qu'il n'a pas déjà été choisi. Pour cela, on stocke les pixels choisis dans un dictionnaire dont la clé est le pixel.

Cependant on ne peut pas placer une liste comme clé d'un dictionnaire. Une possibilité est de convertir la liste en chaîne de caractères avec `str()`.

```
def initialise(img,k):
    n , m = len(img) , len(img[0])
    L=[]
    while len(L)<k:
        i,j=randint(0,n-1),randint(0,m-1)
        if img[i][j] not in L : # vérifier qu'on n'a pas déjà tiré ce
   pixel
            L.append(img[i][j]) # on ajoute le pixel à tab
    return np.array(tab)
```

R4. Écrire une fonction `barycentre(img,s)` qui prend pour argument une image `img` et un ensemble `s` de coordonnées (x, y) et renvoie un pixel (c'est-à-dire un triplet $[r, g, b]$) égal au barycentre (en terme de couleurs) des pixels de l'image dont les coordonnées appartiennent à s.

Solution:

```
def barycentre(img,s):
    bar = [0,0,0]
    for i in range(3):
        for j in range(len(s)):
            x , y = s
            bar[i] = bar[i] + img[x][y][i]
        bar[i] = bar[i]//len(s) # il faut des entiers
    return bar
def barycentre(img,s):
    bar = [0,0,0]
    for i in range(3):
        for pi in s:
            bar[i] = bar[i] + img[pi[0]][pi[1]]
        bar[i] = bar[i]//len(s) # il faut des entiers
    return bar
```

R5. Écrire une fonction `PlusProchePixel(p,mu)` qui prend pour argument un pixel `p` (c'est-à-dire une liste `[r,g,b]` et une liste `mu` de pixels $[\mu_0, ..., \mu_{k-1}]$ et qui renvoie l'indice j qui minimise la distance $\|p - \mu_j\|$.

Solution:

```
def PlusProchePixel(p,mu):
    dmin=dist(p,mu[0])
    jmin=0
    for j in range(1,len(mu)):
        d=dist(p,mu[j]) # distance entre le pixel p et le barycentre j
        if d<dmin:
            dmin=d
            jmin=j
    return jmin
def PlusProchePixel(p,mu):
```

```
    D = [] # liste des distances entre p et mu[i]
    k = len(mu)
    for i in range(k):
        D.append(dist(p,mu[i]))
    jmin = 0
    for j in range(1,k):
        if D[j]<D[jmin]:
            jmin = j
    return jmin
```

R6. En déduire une fonction `kmoyennes(img, k)` qui prend pour arguments une image `img` et un entier `k` et qui renvoie un tableau `s` de longueur `k`, chacun de ses éléments étant un ensemble de coordonnées (x, y) des pixels obtenu par l'algorithme des k-moyennes.

Solution:

```
def kmoyennes(img,k):
    n , m = len(img) , len(img[0])
    mu = initialise(img,k)
    test = True # variable qui passe à False si ça n'évolue plus
    while test: # tant que ça évolue
        s=[[] for i in range(k)] # on crée k classes
        for i in range(n):
            for j in range(m):
                # pour chaque point de l'image
                pi=img[i][j] # pixel correspondant
                ppp=PlusProchePixel(pi,mu) # on cherche le rang du
   pixel dans mu le plus proche
                s[ppp].append([i,j])
        new_mu=[]  # barycentres des nouvelles classes
        for i in range(k):
            new_mu.append(barycentre(img,s[i]))
        if np.array(new_mu).all()==np.array(mu).all():
            test = False # pas d'évolution
        else:
            test = True # ligne inutile !
            mu=deepcopy(new_mu) # copie en profondeur dans mu
    return s
```

Une fois la partition obtenue, il reste à calculer la couleur moyenne de chacune des classes et attribuer cette couleur à chacun des pixels de la classe.

R7. Rédiger une fonction `reduire(img, k)` qui prend pour argument une image et renvoie une nouvelle image dans laquelle seules `k` couleurs sont utilisées.

Solution:

```
def reduire(img,k):
    img2 = [[0 for i in range(len(img[0]))] for j in range(len(img))] #
   future nouvelle image ; tableau de 0 de la taille de img
    s = kmoyennes(img,k) # k classes de couleurs
    for i in range(k): # parcours des k classes
```

```
        coul_moy = barycentre(img,s[i]) # couleur moyenne de la partie s[i]
        for j in range(len(s[i])): # parcours des points de cette classe on attribue la couleur moyenne à tous les points de cette classe
            x,y = s[i][j]
            img2[x][y] = coul_moy
    return img2
```