Institut National des Télécommunications et des Technologies de l'Information et de la Communication INT-TIC d'Oran Algérie

Analyse de Données

3ème année ingénieur

Fiche de TP N° 2

Structuration des Données et Calcul Matriciel

En analyse de données, les données sont souvent structurées sous forme de vecteurs et de matrices (tableaux multidimensionnels). N'étant pas très adaptées au calcul scientifique, les collections du langage Python (voir Annexe de ce TP), notamment les listes, ne permettent pas une manipulation simplifiée des vecteurs et des matrices.

La bibliothèque NumPy apporte une solution à ce problème. Elle propose des structures de données représentant des matrices ou tableaux multidimensionnels. En plus d'être plus simple à manipuler, ces structures sont plus performantes que les collections de base de Python. Un grand nombre de fonctions de manipulations diverses (tri, recherche, extraction, .. etc.) et de calcul (mathématique, statistique, .. etc.) sont fournis par la bibliothèque.

NumPy est à la base d'autres bibliothèques de calcul scientifique telle que SciPy.

Pour l'installation de NumPy, référez-vous à https://numpy.org/ ou https://www.scipy.org/install.html.

Remarque : contrairement aux listes de Python, les tableaux NumPy ne gèrent pas des données de types différents.

1. Création de tableaux multidimensinnels

```
# impoter le module
>>> import numpy as np

# Création de vecteurs et de matrices
>>> v1 = np.array ( [1.5, 5.3, 25.5, 9] )
>>> v2 = np.array ( [1, 3, 2])
>>> v3 = np.array ( [1, 3, 2], dtype=float ) # Création en précisant le type de base
>>> m  = np.array ( [[3.2,2.5],[1.2,4.8],[1.5,6.3]] )

# Affichage
>>> print(v1)
[1.5 5.3 25.5 9]
>>> print(m)
[[3.2 2.5]
 [1.2 4.8]
 [1.5 6.3]]
```

```
# Propriétés
>>> type(v1); type(m)
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
>>> v1.dtype; v2.dtype; v3.dtype; m.dtype      # type de base
dtype('float64')
dtype('int32')
dtype('float64')
dtype('float64')
>>> v1.ndim; v2.ndim ; m.ndim                  # dimension
1
1
2
>>> v1.shape; v2.shape ; m.shape         # nombre d'éléments sur chaque dimension
(4,)
(3,)
(3, 2)
>>> v1.size; v2.size; m.size             # taille : nombre d'éléments
4
3
6

# Création d'un tableau d'objets
>>> v = np.array([{"Alger":(16,3000000)},{"Oran":(31,2000000)}])
>>> v.dtype
dtype('O')

# Création avec la fonction arange
>>> v1 = np.arange(start=0,stop=10)            # dernière valeur n'est pas incluse
>>> print(v1);
[0 1 2 3 4 5 6 7 8 9]
>>> v2 = np.arange(start=0,stop=10,step=2)
>>> print(v2)
[0 2 4 6 8]
>>> m = np.arange(0,10).reshape(2,5)
[[0 1 2 3 4]
 [5 6 7 8 9]]
>>> m = v1.reshape(2,5)                        # même résultat
[[0 1 2 3 4]
 [5 6 7 8 9]]

# Création avec la fonction linspace
>>> v = np.linspace(start=0,stop=10,num=5)
>>> print(v)                      # Ici la dernière valeur est incluse
[0. 2.5 5. 7.5 10.]

# Création de tableaux à valeurs identiques
>>> v1 = np.ones(shape=3)
>>> print(v1)
[1. 1. 1.]
>>> v2 = np.zeros(shape=3)
[0. 0. 0.]
>>> v2 = np.full(shape=3,fill_value=1.2)
print(v2)
[1.2 1.2 1.2]
>>> m = np.full(shape=(2,4),fill_value=0.1)
>>> print(m)
[[0.1 0.1 0.1 0.1]
 [0.1 0.1 0.1 0.1]]

# Création à partir d'une liste python
>>> lst = [1,2,3,4,5,6]
>>> v = np.asarray(lst,dtype=float)
>>> m = np.asarray(lst,dtype=float).reshape(2,3)
>>> print(v); print(m)
[1 2 3 4 5 6]
[[1 2 3]
 [4 5 6]]
```

```
# Création à partir d'un fichier texte
>>> v = np.loadtxt("vecteur.txt",dtype=float)
>>> m = np.loadtxt("matrice.txt",delimiter="\t",dtype=float)
```

2. Ajout, suppression, insertion et redimensionnement

```
# Fonctions "append" et "delete"
>>> v1 = np.array ( [1.5, 5.3, 25.5, 9] )
>>> v2 = np.array ( [11, 12] )
>>> v = np.append(v1,10)          # Ajout d'un élément en dernière position
>>> print(v)
[1.5 5.3 25.5 9. 10.]
v = np.append(v1,v2)              # Concaténation de 2 vecteurs
>>> print(v)
[1.5 5.3 25.5 9. 10. 11. 12.]
>>> v = np.delete(v,2)            # Suppression suivant l'indice
>>> print(v)
[1.5 5.3 9. 10. 11. 12.]

# Redimensionnement d'un vecteur
>>> v = np.array([1,2,3])
>>> v.resize(new_shape=5)
>>> print(v)                      # les nouvelles cases remplies de 0
[1 2 3 0 0]

# Ajout de lignes et de colonnes dans une matrice
>>> m = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
>>> v1 = np.array([[4.1,2.6]])
>>> m1 = np.append(m,v1,axis=0)         #ajouter une ligne
>>> print(m1)
[[1.2 2.5]
 [3.2 1.8]
 [1.1 4.3]
 [4.1 2.6]]
>>> v2 = np.array([[7.8],[6.1],[5.4]])  #ajouter une colonne
>>> m2 = np.append(m,v2,axis=1)
>>> print(m2)
[[1.2 2.5 7.8]
 [3.2 1.8 6.1]
 [1.1 4.3 5.4]]

# Insertion à un endroit défini
>>> print(np.insert(m,1,v1,axis=0))
[[1.2 2.5]
 [4.1 2.6]
 [3.2 1.8]
 [1.1 4.3]]

# Suppression d'un endroit défini
>>> print(np.delete(m,1,axis=0))
[[1.2 2.5]
 [1.1 4.3]]

# Redimensionnement de la matrice
>>> h = np.resize(m,new_shape=(2,3))
print(h)
[[1.2 2.5 3.2]
 [1.8 1.1 4.3]]
```

3. Extraction de plages de valeurs (sous-tableaux)

```
# Pour un vecteur
>>> v = np.array([7,2.3,5.2,3.6,1.9])
>>> print(v[:])            # tout les éléments (tout comme print(v))
[7. 2.3 5.2 3.6 1.9]
>>> print(v[0])            # Accès avec indice
7.
>>> print(v[-1])           # dernier élément (tout comme print(v[v.size-1]))
1.9
>>> print(v[1:3])          # plage d'indices
[2.3 5.2]
>>> print(v[:3])           # du début l'élément d'indice 3 (non inclus)
[7. 2.3 5.2]
>>> print(v[2:])           # du 2ème élément jusqu'à la fin
[5.2 3.6 1.9]
>>> print(v[-3:])          # 3 derniers éléments
[5.2 3.6 1.9]
>>> print(v[1:4:2])        # avec un pas de 2
[2.3 3.6]
>>> print(v[3:0:-1])       # avec un pas de -1
[2.3 3.6]
print(v[::-1])             # pour avoir un vecteur inversé
[1.9 3.6 5.2 2.3 7.]

# Pour une matrice
>>> m = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
>>> print(m[:,:])          # tout les éléments (tout comme print(v))
[[1.2 2.5]
 [3.2 1.8]
 [1.1 4.3]]
>>> print(m[0,0])          # Accès avec indice
1.2
>>> print(m[-1,-1])        # dernier élément (tout comme print(m[m.shape[0]-
1,m.shape[1]-1]))
4.3
>>> print(m[0:2,:])        # plage d'indices
[[1.2 2.5]
 [3.2 1.8]]
>>> print(m[:2,:])         # même résultat que le précédent
[[1.2 2.5]
 [3.2 1.8]]
>>> print(m[1:,:])
[[3.2 1.8]
 [1.1 4.3]]
>>> print(m[-1,:])         # dernière ligne (toutes les colonnes)
[[1.1 4.3]]
>>> print(m[-2:,:])        # 2 dernières lignes
[[3.2 1.8]
 [1.1 4.3]]

# Extraction en utilisant des conditions
>>> b = v < 5                                  # Cette opération renvoie un tableau de
booléens
>>> print (b)
[False True False True True]
>>> print(v[b]); print(v[v<7])          # les deux instructions sont équivalentes
[2.3 3.6 1.9]
>>> b = np.array([True,False,True],dtype=bool)       # création d'un vecteur de
booléens
>>> print(m[b,:])                              # extraction suivant le vecteur booléen
[[1.2 2.5]
 [1.1 4.3]]
>>> print(m[m[:,0]<5,:])               # même résultat (m[:,0] est la colonne
d'indice 0)
[[1.2 2.5]
 [1.1 4.3]]
```

Exercice : Extraire de la matrice `m` les lignes dont la somme est égale au minimum.

Dans un premier temps, on calcule la somme des lignes :

```
>>> s = np.sum(m,axis=1)
>>> print(s)
[ 3.7 5. 5.4 ]
```

On peut calculer le minimum du vecteur s :

```
>>> np.min(s)
3.7
```

Le vecteur booléen b permet de repérer les lignes correspondant au minimum :

```
>>> b = (s == np.min(s))
>>> print(b)
[True False False]
```

Il ne reste plus qu'à appliquer le filtre booléen :

```
>>> print(v[b,:])
[[1.2 2.5]]
```

4. Tri et Recherche

```
# Recherche du min et du max
>>> v = np.array([7,2.3,5.2,3.6,1.9])
>>> m = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
>>> print(np.max(v))              # Max d'un vecteur (min fonctionne de la même façon)
7.
>>> print(np.argmax(v))           # Indice du Max (argmin fonctionne de la même façon)
0
>>> print(np.max(m,axis=0))       # Max de chaque colonne
[3.2 4.3]
>>> print(np.max(m,axis=1))       # Max de chaque ligne
[2.5 3.2 4.3]
>>> print(np.argmax(m,axis=0))    # Indice des Max
[1 2]

# Recherche d'une valeur donnée
>>> print(np.where(v==3.6))          # Indices des éléments du vecteur égaux à
3.6
(array([3], dtype=int64),)
>>> print(np.where((m>2)&(m<4))      # Indices des éléments de la matrice >2
et <4
(array([0,1], dtype=int64),array([1,0], dtype=int64))


# Tri
>>> print(np.sort(v))             # Tri d'un vecteur
[1.9 2.3 3.6 5.2 7]
>>> print(np.argsort(v))          # Indices des éléments triés
[4 1 3 2 0]
>>> print(np.sort(m,axis=0))      # Tri d'une matrice selon une dimension
[[1.1 1.8]
 [1.2 2.5]
 [3.2 4.3]]
>>> print(np.argsort(m,axis=0))   # Indices des éléments triés
[[2 1]
 [0 0]
 [1 2]]
>>> a = np.array([1,2,2,1,1,2])   # Eléments uniques (sans doublons)
>>> print(np.unique(a))
[1 2]
```

5. Opérations ensemblistes

```
v1 = np.array([1,2,5,6])
V2 = np.array([2,1,7,4])
>>> print(np.intersect1d(x,y))    # intersection
[1 2]
>>> print(np.union1d(x,y))              # union (à ne pas confendre avec une
concaténation)
[1 2 4 5 6 7]
>>> print(np.setdiff1d(x,y))      # différence
[5 6]
```

6. Calcul matriciel

```
# Calculs entre vecteurs : élément par élément
>>> v1 = np.array([3.2,5.1,2.5])
>>> v2 = np.array([8.4,0.7,4.6])
>>> print(v1+v2)                  # addition
[11.6 5.8 7.1]
>>> print(v1*v2)                  # multiplication
[26.88 3.57 11.5]
>>> print(2*v1)                   # multiplication par un scalaire
[6.4 10.2 5.]
>>> b = v1 > v2                   # comparaison
>> print(b)
[False True False]

# Opérations logiques entre vecteurs de booléens
>>> a = np.array([True,True,False,True],dtype=bool)
>>> b = np.array([True,False,True,False],dtype=bool)
>>> print(np.logical_and(a,b))          # ET logique
[True False False False]
>>> print(np.logical_or(a,b))           # OU logique
[True True True True]
>>> print(np.logical_xor(a,b))          # OU exclusif logique
[False True True True]


# Norme et Produit scalaire
>>> v1 = np.array([1.2,1.3,1.0])
>>> v2 = np.array([2.1,0.8,1.3])
>>> p = np.vdot(v1,v2)            # produit scalaire
>>> print(p)
4.86
>>> print(np.sum(v1*v2))   # autre façon de calcul du produit scalaire
4.86
>>> n = np.linalg.norm(v1)# norme d'un vecteur
>>> print(n)
2.03
>>> import math                         # autre façon de calcul de la norme
>>> print(math.sqrt(np.sum(v1**2)))
2.03

# Transposé, Produit matriciel, Déterminant et Matrice inverse
>>> m1 = np.array([[1.2,2.5],[3.2,1.8],[1.1,4.3]])
>>> m2 = np.array([[2.1,0.8],[1.3,2.5]])
>>> print(np.transpose(m1))             # transposé
[[1.2 3.2 1.1]
 [2.5 1.8 4.3]]
>>> print(np.dot(m1,m2))          # produit matriciel
[[5.77 7.21]
 [9.06 7.06]
 [7.9 11.63]]
>>> print(np.linalg.det(m2))      # déterminant
4.21
```

```
>>> print(np.linalg.inv(m2))      # inversion
[[ 0.59382423 -0.19002375]
 [-0.3087886   0.49881235]]

# Résolution d'équation
>>> M = np.array([[2.1,0.8],[1.3,2.5]])
>>> y = np.array([1.7,1.0])
>>> print(np.linalg.solve(M,y))         # trouver x tel que M.x = y
[0.8195 -0.0261]
>>> print(np.dot(np.linalg.inv(M),y))   # pour vérifier : x = inverse(M).y
[0.8195 -0.0261]

# Matrice symétrique avec transposé(M).M
>>> S = np.dot(np.transpose(M),M)
>>> print(S)
[[6.1  4.93]
 [4.93 6.89]]

# Valeurs et vecteurs propres d'une matrice symétrique
>>> print(np.linalg.eigh(S))
(array([1.54920128, 11.44079872]), array([[-0.73480125, 0.67828248], [0.67828248,
73480125]]))
```

R. HACHEMI