Feuille d'exercices 9
Apprentissage supervisé / non supervisé

Afin de tester nos algorithmes d'apprentissage, nous allons utiliser la bibliothèque scikit-learn de Python, qui met à disposition algorithmes et outils pour la fouille de données, mais surtout en ce qui nous concerne, dans son sous-module datasets, des ensemble de données permettant de tester ces algorithmes.

La documentation est accessible à : `https://scikit-learn.org/stable/index.html`

Les ensembles de données à : `https://scikit-learn.org/stable/datasets/toy_dataset.html`

Nous nous focaliserons sur l'ensemble de données digits qui contient des images de chiffres écrits à la main :

1797 images sous forme de tableau bitmaps 8×8 en niveau de gris d'intensité dans $[[0, 16]]$.

```
from sklearn.datasets import load_digits
digits = load_digits()
```

la variable `digits` contiendra alors l'ensemble des données. Le tableau des pixels des images s'obtient grâce à l'attribut `data` :

```
In [1]: digits.data
Out[1]:
array([[  0.,   0.,   5., ...,   0.,   0.,   0.],
       [  0.,   0.,   0., ...,  10.,   0.,   0.],
       [  0.,   0.,   0., ...,  16.,   9.,   0.],
       ...,
       [  0.,   0.,   1., ...,   6.,   0.,   0.],
       [  0.,   0.,   2., ...,  12.,   0.,   0.],
       [  0.,   0.,  10., ...,  12.,   1.,   0.]])

In [2]: digits.data.shape
Out[2]: (1797, 64)
```

C'est un tableau numpy bidimensionnel de taille 1797 (nbre images) $\times 64$ (8×8 bits par image).
On peut afficher quelques images à l'aide du code :

```
import matplotlib.pyplot as plt
plt.gray()
for k in range(20):
   plt.figure(k)
   plt.matshow(digits.images[k])
   plt.show()
```

et l'étiquetage se trouve dans l'attribut `target` qui est un tableau unidimensionnel de 1797 entiers.

```
In [3]: digits.target[:20]
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [4]: digits.target.shape
Out[4]: (1797,)
```

Par exemple, le première image est représentée par le tableau (unidimensionnel) de 64 pixels :

```
In [5]: digits.data[0]
Out[5]: array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,
                 0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.,
                 0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.,
                 0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,
                 0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.,
                 0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.,
                 0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.,
                 0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Exercice 1 Algorithme des k voisins

1. Écrire une fonction `dist` prenant en argument deux tableaux de pixels et renvoyant leur distance (on pourra utiliser la fonction `norm` de `numpy.linalg` pour la distance euclidienne).
2. Pour $N = 200$: énumérer au sein d'une liste de 10 éléments le nombre de 0, 1, 2,..., 9 parmi les N premiers éléments de la base, puis l'afficher afin de vérifier qu'ils sont environ en même nombre.
3. Constituer une base d'apprentissage de N éléments, constitués d'une liste `Base` de couples `(T,n)` où `T` est un tableau de pixels, et `n` son étiquette.
4. Écrire une fonction `kPlusProchesVoisins(Base,k,T)` prenant en argument la base d'apprentissage, l'entier k et un tableau de 64 pixels (qu'on choisira aléatoirement dans `digits.data`) et qui renverra une estimation de l'étiquette de `T` par l'algorithme des k plus proches voisins. L'essayer sur plusieurs tableaux de pixels
5. Écrire une fonctions `matriceConfusion(Base,k,n)` qui donne la matrice de confusion de la fonction `kPlusProchesVoisins(Base,k,T)` construite en tirant au sort n tableaux de pixels dans `digits.data`. Elle renverra aussi pour plus de lisibilité la proportion d'éléments sur la diagonale. La tester pour plusieurs valeurs de n et de k.

Exercice 2 Algorithme des k-moyennes

On va appliquer à ces données un apprentissage non supervisé afin de constituer 10 classes qui, on l'espère, correspondront le mieux possible aux 10 chiffres cardinaux. On va appliquer pour cela l'algorithmes des k-moyennes, avec ici $k = 10$.

```
from sklearn.datasets import load_digits
digits = load_digits()

from numpy import array
from numpy.linalg import norm

def dist(T1,T2):
    return norm(T1-T2)

n = 200 # Limite au nombre d'images ; à augmenter
Images = digits.data[:n]
Etiquettes = digits.target[:n]
N = len(Images)
```

Afin d'accélerer le temps de calcul on se limite aux $n = 200$ premières données ; on pourra ensuite augmenter la valeur de n. Ainsi `Image` est la liste des $n = 200$ premiers tableaux de pixels (des nd.arrays de 64 éléments dans $[[0, 16]]$), et `Etiquettes` est la liste de leurs étiquettes respectives, 0,1,...,9. On a pris pour distance, la distance euclidienne de $\mathbb{R}^{64}$ obtenue grâce à la fonction `norm`.

Implémentation : les 10 classes seront les listes des indices des objets choisis ; donc des entiers entre 0 et $n-1$. Cela permettra de retrouver ensuite leurs étiquettes pour tester la fiabilité de notre approche.

1. Fonction `barycentre`.

   Écrire une fonction `barycentre(Ci)` prenant en entrée une liste `Ci` d'indices dans $[[0, n-1]]$, et qui renvoie un nd.array représentant le tableau de 64 pixels moyen (isobarycentre) des tableaux de pixels des images dans `Images` représentés par la liste d'indice.

   Ainsi la fonction prend en entrée une liste d'indice de vecteurs de $\mathbb{R}^{64}$ et renvoie l'isobarycentre de ces vecteurs.

2. Algorithme des k-moyennes.

   Adapter l'algorithme des k-moyennes vu en cours au contexte présent pour qu'il renvoie une liste `C` des 10 classes, chacune étant une liste d'entiers compris entre 0 et $n-1$.

3. Phase de test.

   Ecrire un script qui pour chacune des 10 classes, affiche l'étiquette majoritaire et la proportion d'éléments de la classe ayant cette étiquette.