Pour ce TP, commencez par télécharger le fichier TP_IA.zip sur la page web du cours. Dézipper le fichier (clic droit → extraire ici), et ouvrir le fichier TP_IA.pyavec Pyzo. Ensuite, modifier l'adresse du dossier TP_IA/ dans la commande `os.chdir` pour qu'elle corresponde à l'adresse de votre dossier sur votre ordinateur. Exécuter le fichier :

- si le message vous dit qu'il n'y a pas de problème, commencer le TP ;
- si le message vous dit qu'il y a une erreur, m'appeler.

Dans le dossier TP_IA/se trouvent 200 images de chiffres sous la forme `Image_xxx_y.png`, où `xxx` est un nombre entre `000` et `199`, et `y` est un chiffre entre `0` et `9` (le chiffre représenté par l'image). Ce dossier contient également quelques images en couleur pour la fin du TP.
Le fichier TP_IA.py fournit les fonctions suivantes :

- `image_assoc(nom_image)` renvoie une représentation de l'image `nom_image` sous la forme d'un tableau Numpy ;
- `chiffre_assoc(nom_image)` renvoie le chiffre `y` d'une image décrite précédemment ;
- `recupere_image()` renvoie une liste de 200 couples de la forme `(im,c)`, où `im` est une image de chiffre importée au format Numpy, et `c` est le chiffre représenté par cette image ;
- `afficher_image(im)` affiche l'image `im` (si `im` est sous la forme d'un tableau Numpy obtenu via les fonctions précédentes).

Rappel sur les tableaux Numpy et les images. Une image de taille $h \times \ell$ est représentée en Python par un tableau Numpy de dimension $h \times \ell$.

- Si `im` est un tableau Numpy représentant une image en noir et blanc :
  - l'instruction `h,l = im.shape` récupère les dimensions de l'image ;
  - une case `im[i][j]` contient une valeur dans $[\![0, 255]\!]$ représentant le niveau de gris du pixel (i, j) (0 signifie noir, 255 siginifie blanc).
- Si `im` est un tableau Numpy représentant une image en couleur :
  - l'instruction `h,l,p = im.shape` récupère les dimensions de l'image (la troisième valeur `p` indique le nombre de couleurs primaires utilisées : normalement, `p` vaut 3) ;
  - une case `im[i][j]` contient un tableau Numpy de longueur 3 de valeurs dans $[\![0, 255]\!]$ représentant le niveau de rouge (resp. vert, bleu) du pixel (i, j).

On rappelle qu'on peut créer un tableau Numpy rempli de zéros à l'aide des instructions suivantes :

```
T = np.zeros((h,l), dtype=int) # les cases sont des entiers

T = np.zeros((h,l), dtype=float) # les cases sont des flottants
```

Pour choisir entre le type `int` et le type `float`, on fera comme suit :

- si `T` représente une image qu'on voudra afficher, ou si l'on est sur que les valeurs stockées seront toujours entières (comme dans l'algorithme des k plus proches voisins), il faut utiliser le type `int` ;
- si `T` va stocker des résultats non entiers, comme dans l'algorithme des k-moyennes, il faut utiliser le type `float`.

On rappelle également qu'on peut convertir une liste `L` en tableau Numpy à l'aide de l'instruction suivante :

```
T = np.array(L)
```

1 Algorithme des k plus proches voisins

Le but de cette partie est d'utiliser l'algorithme des k plus proches voisins pour identifier le chiffre représenté par une image. Dans cette partie, on utilisera les tableaux suivants :

```
Z = recupere_image()
X = Z[:150] # ensemble d'apprentissage
Y = Z[150:] # ensemble de test
```

Distance entre deux images. On va utiliser la distance suivante pour deux images en noir et blanc de même taille $h \times \ell$:

$$d(I, I') = \sum_{i=0}^{h-1} \sum_{j=0}^{\ell-1} |I_{i,j} - I'_{i,j}|$$

1. Écrire une fonction `distance(im1,im2)` renvoyant la distance entre deux images.

Exemple

```
>>> distance(Z[0][0], Z[1][0])
29506
```

2. Écrire une fonction `k_voisins(X,k,im)` prenant en entrée :
   - une liste `X` constituée de couples `(im2,c)` où `c` est le chiffre représentée par l'image `im2` ;
   - un entier $k <$ `len(X)` ;
   - une image `im` ;

   et renvoyant le chiffre majoritaire parmi les k plus proches voisins de `im` dans `X`.

Exemple

```
>>> k_voisins(X,10,Z[151][0]) # bonne réponse
6
>>> k_voisins(X,10,Z[150][0]) # la bonne réponse était 4
1
```

Matrice de confusion. Pour décider quel valeur de k est optimale pour notre algorithme, on va utiliser l'ensemble de test `Y`, et regarder sur combien d'exemples on se trompe. Pour cela, on va construire la **matrice de confusion** C de taille 10×10 dont les coefficients sont les suivants :

$$c_{i,j} = \text{nombre d'éléments de } \texttt{Y} \text{ dont le chiffre est } i \text{ mais où l'algorithme a attribué le chiffre } j$$

Plus la matrice obtenue se rapproche d'une matrice diagonale, plus le choix de k utilisé est pertinent.

3. Écrire une fonction `matrice_confusion(X,Y,k)` renvoyant la matrice de confusion associée à ses arguments.

Exemple

```
>>> matrice_confusion(X,Y,5)
array([[3, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 4, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 2, 5, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 4, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 4, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 3, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 3, 0, 0, 0],
       [0, 2, 0, 0, 0, 0, 0, 3, 0, 0],
       [0, 1, 0, 0, 1, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 0, 3, 0, 6]])
```

4. Écrire une fonction `dist_diag(C)` calculant la somme des coefficients de C (sauf ceux de la diagonale).

Exemple

```
>>> dist_diag(matrice_confusion(X,Y,5))
14
```

5. Écrire une fonction `meilleur_k(X,Y,k_max)` calculant les matrices de confusions C pour toutes les valeurs $k \in [\![1, k_{\max}]\!]$, et renvoyant la valeur de k qui minimise `dist_diag(C)`.
   Tester la valeur obtenue pour `meilleur_k(X,Y,12)`.

2 Algorithme des k moyennes

Le but de cette partie est de partitionner l'ensemble des 200 images de chiffres en 10 catégories à l'aide de l'algorithme des k moyennes.

6. Écrire une fonction `barycentre(C)` prenant en argument une liste `C` d'images, et renvoyant le barycentre de ces images.
7. Écrire une fonction `k_moyennes(X,k)` implémentant l'algorithme des k-moyennes, et renvoyant un couple `(mu, C)` où `mu` est le tableau des barycentres obtenus, et `C` est la liste des k classes correspondantes.
   Remarque : on utilisera des tableaux Numpy pour stocker `mu` et `new_mu`, et il faudra alors utiliser l'instruction suivante pour tester l'égalité : `if (new_mu == mu).all():`

On pourra afficher les barycentres obtenus à l'aide des instructions suivantes :

Exemple

```
mu, C = k_moyennes(Z,10)

for im in mu:
    afficher_image(im)
```

3 Application : compression d'images

Dans cette partie, nous travaillons avec des images en couleur. Le but de cette partie est de compresser une image en couleur, en diminuant considérablement le nombre de couleurs différentes utilisées.

On pourra utiliser la photo de la tour Eiffel fournie :

```
tour_eiffel = image_assoc("tour_eiffel.png")

afficher_image(tour_eiffel)
```

8. Écrire une fonction `nombre_couleurs(im)` qui calcule le nombre de couleurs différentes utilisées dans l'image `im`.

Exemple

```
>>> nombre_couleurs(tour_eiffel)
133309
```

Notre objectif est de réduire ce nombre à seulement 16 couleurs, tout en gardant une image la plus fidèle possible à l'image de départ. Pour cela, nous allons utiliser l'algorithme des k moyennes pour classer les différents pixels de l'image en 16 classes : les barycentres correspondants nous donneront alors les 16 couleurs à utiliser.
Une fois ce calcul effectué, il restera alors à attribuer la couleur μ_i à chaque pixel de la classe C_i.

Rappel. Dans une image en couleur, chaque pixel est un vecteur de taille 3 (représentant les nuances de rouge/vert/bleu du pixel).

9. Écrire une fonction `dist(a,b)` prenant en arguments deux vecteurs de taille 3 (sous la forme de tableaux Numpy) et renvoyant $||a-b||_2$.

Exemple

```
>>> dist(tour_eiffel[12][12],tour_eiffel[42][42])
37.376463182061514
```

10. Écrire une fonction `init(im,k)` renvoyant un tableau Numpy `mu` de longueur k contenant k pixels de l'image `im` tirés au hasard.
Remarque : on pourra importer la librairie `random` et utiliser la fonction `rd.randrange(n)` qui tire au hasard un entier dans $[\![0, n-1]\!]$.

Exemple

```
import random as rd

i = rd.randrange(10) # i est tiré au hasard entre 0 et 9
```

11. Écrire une fonction `bary(im,L)` prenant en arguments une image en couleur (sous forme d'un tableau Numpy), et une liste de coordonnées de pixels, et renvoie le barycentre de ces pixels (sous la forme d'un tableau Numpy de taille 3).

Exemple

```
>>> bary(tour_eiffel, [(0,0), (12,42), (200,150)])
array([ 23.        , 119.33333333, 219.66666667])
```

12. Écrire une fonction `plusProchePixel(p,mu)` prenant en arguments un pixel `p` (tableau Numpy de taille 3) et une liste de pixels $[\mu_0, \dots, \mu_{k-1}]$, et renvoyant l'indice j qui minimise $||p-m_j||_2$.
13. Écrire une fonction `kmoyennes(im,k)` prenant en arguments une image `im` et une valeur k, et choisit k couleurs en utilisant l'algorithme des k moyennes.
On renverra la tableau des barycentres `mu` ainsi que les classes de coordonnées de pixels `C`.
Remarque : pour que l'algorithme s'arrête avant la fin du TP, on pourra remplacer le test d'arrêt `if (new_mu == mu).all(): ...` par `if np.linalg.norm(new_mu-mu) < k: ...`
14. Écrire une fonction `compresser(im,k)` qui calcule les valeurs `mu` et `C` via la fonction précédente, et renvoie une nouvelle image où chaque pixel de coordonnées (x,y) dans C_i contient la couleur du pixel μ_i.

On pourra tester le résultat obtenu via ces instructions :

```
tour_eiffel_comp = compresser(tour_eiffel,16)

afficher_image(tour_eiffel_comp)
```