TD : RECONNAISSANCE DE CARACTÈRES AVEC KNN

(TD inspiré de CCINP-PSI-Info 2023)

La reconnaissance optique de caractères (OCR) existe depuis de nombreuses années mais les récents travaux d'intelligence artificielle (apprentissage profond) ont considérablement augmenté les performances de la reconnaissance de documents.

L'objectif du travail proposé dans le sujet original est de découvrir différentes étapes de la numérisation d'un document en explorant plusieurs algorithmes utilisés pour obtenir finalement un document éditable conforme à l'original.

Les parties abordées dans le sujet sont les suivantes :

- Partie 1 : Acquisition d'un document et pré-traitement dans le but d'obtenir une image numérique pertinente ;
- Partie 2 : Reconnaissance du contenu qui correspond à l'extraction du texte et de sa structure ;
- Partie 3 : Reconnaissance des caractères par identification à l'aide d'une base de données.

La reconnaissance de caractères est une chaîne de traitement complète dont le KNN est l'étape finale. Ce TD se concentre sur la reconnaissance des caractères par identification développée dans la partie 3. Voici un résumé des parties qui la précèdent :

I) PARTIE 1 : ACQUISITION D'UN DOCUMENT

Le document est numérisé. Pour diminuer la taille du document afin de pouvoir plus facilement le traiter, on réalise tout d'abord une conversion en niveaux de gris de l'image. La formule utilisée pour déterminer la valeur d'un pixel gris en fonction des trois couleurs d'un pixel (R rouge, G vert, B bleu) est la suivante :

$$pixGris = 0{,}299 \cdot R + 0{,}587 \cdot G + 0{,}114 \cdot B$$

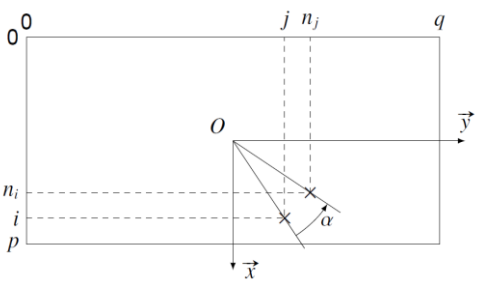
On applique ensuite une binarisation (seuil) pour n'avoir que du noir (0) et du blanc (255). La difficulté de cette technique de binarisation est le choix de la valeur seuil pour des images ayant des problèmes d'éclairage. Le sujet aborde ultérieurement une technique de restauration qui peut être utilisée pour remplacer la binarisation par seuil standard.

II) PARTIE 2 : RECONNAISSANCE DU DOCUMENT

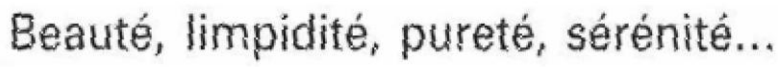
Rotation : L'image scannée peut avoir un problème de rotation qu'il convient de corriger. On redresse donc l'image si elle a été scannée de travers en utilisant une matrice de rotation et une interpolation bilinéaire pour éviter la pixellisation.

Pour faire tourner le point de coordonnées (i, j) autour du point O (centre de l'image) d'un angle α, on applique une rotation à l'aide d'une matrice de rotation :

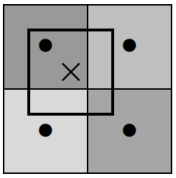
$$\begin{pmatrix} n_i - p/2 \\ n_j - q/2 \end{pmatrix} = \begin{pmatrix} cos\ \alpha & sin\ \alpha \\ -sin\ \alpha & cos\ \alpha \end{pmatrix} \begin{pmatrix} i - p/2 \\ j - q/2 \end{pmatrix}$$



Naïvement, on pourrait penser que pour réaliser la rotation, il suffit de parcourir chaque pixel de l'image initiale en lui appliquant la rotation définie précédemment. Mais les indices étant des entiers, on se rend compte que certains pixels de la nouvelle image ne sont jamais calculés et qu'il peut apparaître des problèmes de dépassement de taille d'image :



L'algorithme de rotation consiste donc, pour chaque pixel de la nouvelle image de coordonnées (ni, nj), à trouver ses coordonnées (i, j) par une rotation d'angle $-\alpha$ dans l'image initiale. La position du pixel virtuel ainsi trouvée est en fait un couple de réels (x, y). Le pixel virtuel est ainsi entouré de 4 pixels dans l'image initiale dont les abscisses sont comprises entre `int(x)` et `int(x)+1` et les ordonnées entre `int(y)` et `int(y)+1`.



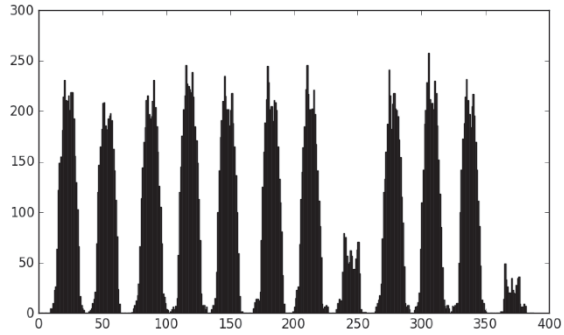
Pour trouver la valeur du pixel virtuel, on utilise la valeur des 4 pixels voisins en réalisant une approximation bilinéaire qui consiste :

- En prenant les deux pixels voisins de la première ligne, à trouver la valeur du niveau de gris du pixel virtuel en supposant une évolution linéaire selon la coordonnée y entre le pixel de gauche et le pixel de droite ;
- À faire de même en prenant les pixels de la deuxième ligne ;
- Enfin en travaillant sur la coordonnée x, à supposer une évolution linéaire entre les deux valeurs trouvées aux deux étapes précédentes.

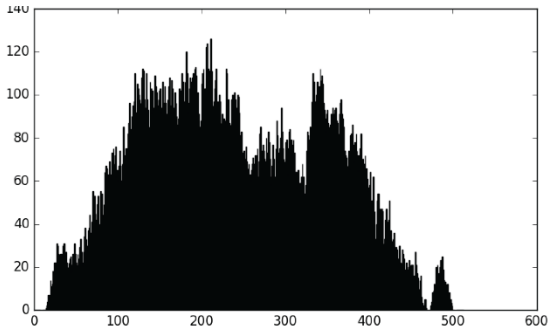
Voici quelques TD disponibles sur le site de https://www.cpge-sii.com/informatique/itc1/ de M. DEFAUCHY sur ces thèmes :

- *TD « 7-2 - Détection de contours – Convolution »*
- *TD « 7-3 – Transformations »*
- *TD « 7-17 - Recadrage projectif »*

Segmentation : Après rotation de l'image, on applique la segmentation. La segmentation consiste à découper l'image en plusieurs éléments de manière à pouvoir ensuite traiter chacun des éléments. Il faut dans l'image pouvoir dissocier les lignes, les mots puis les lettres. L'idée est de construire la liste du nombre de pixels noirs par ligne (histogramme horizontal) pour isoler les lignes de texte, puis par colonne (histogramme vertical) pour isoler chaque caractère.



Histogramme de détection des lignes



Histogramme de détection des lignes sur la figure mal orientée

En appliquant cette détection de ligne directement sur l'image mal orientée, il en résulte une erreur de détection. En effet, si on observe l'histogramme dans ce cas, on constate qu'il n'y a plus de zones avec des pixels blancs détectées. Cela permet de détecter automatiquement la bonne orientation en travaillant sur la maximisation du nombre de 0.

Après avoir séparé les lignes, en appliquant une méthode similaire, on peut extraire les caractères sur chacune de ces lignes :

Restauration : Les images de caractères peuvent être bruitées compte tenu d'une mauvaise résolution ou de parasites apparaissant pendant un scan. De même, la technique de binarisation proposée initialement ne donne pas toujours un résultat correct si le seuil est mal choisi. Le sujet propose alors d'étudier la méthode du flot maximal (ou méthode de la coupe minimale) reposant sur la représentation par un graphe de l'image à restaurer est souvent utilisée pour pallier ces problèmes.

III) PARTIE 3 : DÉTERMINATION DES CARACTÈRES

Une fois les images de lettres isolées, il s'agit de reconnaître la lettre correspondante. Différentes méthodes peuvent être employées. Le sujet étudie une méthode d'apprentissage automatique basée sur les K plus proches voisins.

Le principe de cette méthode consiste à comparer chaque caractère à un ensemble de caractères définis dans une base de données.

On suppose qu'on dispose d'une liste `fichiers_car_ref` contenant les noms des fichiers images d'un grand nombre de caractères ayant des fontes proches de celles du texte scanné. Le nom de chaque fichier est défini de la manière suivante :

```
nomFonte + "_" + nomCatégorie + taillePolice + "_" + idSymbole + ".png"
```

Les catégories sont définies par la liste :

```
categories = ["majuscules","minuscules","chiffres","special"]
```

Les symboles considérés sont définis par la liste :

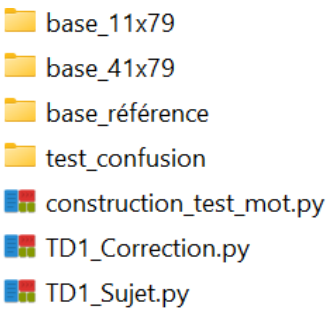
```
symboles = ["ABCDEFGHIJKLMNOPQRSTUVWXYZ","abcdefghijklmnopqrstuvwxyz",
            "0123456789",".:,;'(!?)éèàçùêûâ"].
```

On compte 79 symboles différents. Exemple : `Zurich Light BT_majuscules18_10.png` pour la majuscule K de la police Zurich Light BT en taille 18.

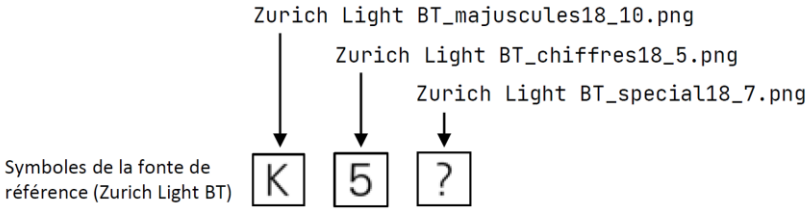
III.1. Installation des ressources pour le TD

Télécharger https://www.informatique-f1.fr/IA/TD2_KNN_dataset.zip et extraire son contenu dans votre répertoire de travail.

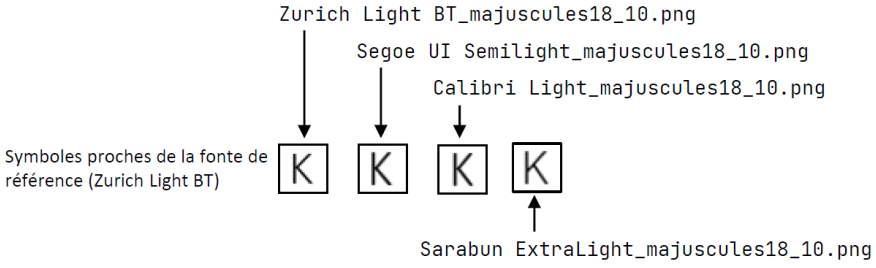
Ce dossier contient l'ensemble des ressources d'images nécessaires pour entraîner et tester un classifieur KNN de reconnaissance de caractères. Il regroupe des bibliothèques contenant des symboles de la police de référence qui serviront à fabriquer des phrases de test et de polices similaires. La taille des images est de 28x28 pixels.



Le dossier « base_référence » contient 79 images (une par symbole) rendues avec la police de référence (Zurich Light BT). Voici quelques exemples :



Le dossier « base_11x79 » contient 11×79 images (les mêmes 79 symboles rendus avec 10 polices supplémentaires visuellement proches de la référence). Voici quelques exemples pour le symbole K :



Le script « construction_test_mot.py » assemble un mot de votre choix en copiant, caractère par caractère, les images déjà présentes dans le dossier « base_référence » vers les dossiers « test_mot », afin que vous puissiez tester le KNN sur le mot que vous souhaitez analyser, et cela avec une dégradation des symboles afin de simuler la numérisation et pré-traitements. Nous utiliserons ce script un peu plus tard dans le TD.

L'objectif final du TD est de mettre en œuvre un classifieur k-plus proches voisins (KNN) pour reconnaître automatiquement des caractères, puis d'évaluer sa robustesse.

Les catégories et les symboles sont définis dans le fichier python de cette manière :

```
# Listes de symboles
categories = ["majuscules", "minuscules", "chiffres", "special"]
symboles = ["ABCDEFGHIJKLMNOPQRSTUVWXYZ","abcdefghijklmnopqrstuvwxyz",
            "0123456789",".:,;'(!?)éèàçùêûâ"]
```

On introduit la fonction suivante :

```
def lire_symbole_fichier(nomFichier: str) -> str:
    car = nomFichier.split('_')
    num = car[2].split('.')[0]
    var = car[1][:len(car[1])-2]
    ind = categories.index(var)
    return symboles[ind][int(num)]
```

On rappelle que pour une liste `L`, `L.index(val)` renvoie la position de `val` dans la liste `L`.

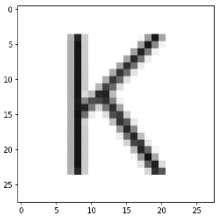
1. Indiquer ce que valent les variables `car`, `num`, `var`, `ind` et ce qui est renvoyé par la fonction si `nomFichier="Zurich Light BT_majuscules18_10.png"`.

Toutes les images des caractères de référence sont lues et stockées sous forme de tableaux `array`. On définit un dictionnaire `carac_ref` dont les clés seront les symboles apparaissant dans la liste symboles (par exemple `"A"`, `"a"`, …). À chaque clé sera associée une liste de tableaux `array` représentant des images.

La commande `img=imread(nomFichier)` permet de lire le fichier image `nomFichier` et de stocker le tableau `array` à deux dimensions qui représente l'image dans la variable `img`.

2. Écrire la fonction `lire_donnees_ref(dossier:str,fichiers_car_ref:list)->dict` qui prend en arguments le dossier et la liste des noms de fichiers images contenus dans ce dossier `fichiers_car_ref` et qui renvoie le dictionnaire contenant tous les tableaux catégorisés.

Tester : `caract_ref=lire_donnees_ref("base_11x79", liste_images_à_lire)`



```
>>> caract_ref.keys()
dict_keys(['K', 'P', ' ?'])
>>> len(caract_ref['K'])
3
>>> imshow(caract_ref['K'][0],cmap="gray",vmin=0,vmax=1)
>>> plt.show()
```

Un caractère à identifier est également stocké sous forme d'un tableau `array` nommé `carac_test`. On suppose que les dimensions de ce tableau et de tous les tableaux du dictionnaire `carac_ref` sont les mêmes.

La méthode d'identification utilisée est celle des K plus proches voisins. Elle consiste à calculer une distance entre l'image du caractère à identifier et toutes les images de référence. En notant (i, j) les coordonnées d'un pixel dans le tableau représentant l'image, p_{ij} le pixel associé à l'image du caractère à identifier et q_{ij} celui d'un caractère de référence, on calcule pour chaque caractère de référence la distance :

$$d = \sqrt{\sum_{i,j} \left(p_{ij} - q_{ij}\right)^2}$$

Les distances d sont stockées dans un dictionnaire `distances` où, pour chaque clé égale à un symbole de la liste `symboles`, on associe une liste de distances pour chaque image de référence de ce symbole.

3. Écrire une fonction `distance(im1:array, im2:array)->float` qui calcule la distance entre les deux images `im1` et `im2` supposées de même dimension.

Le fichier source importe la librairie `numpy` avec la commande : `import numpy as np`. Cette librairie contient une fonction « linalg.norm » dont voici la définition :

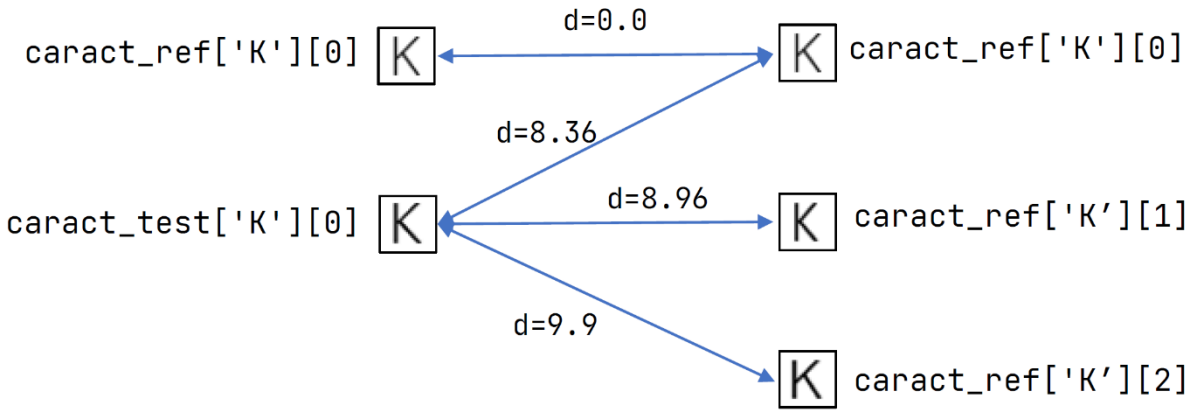
numpy.linalg.norm : `linalg.norm(x, ord=None, axis=None, keepdims=False)`

Matrix or vector norm. This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the ord parameter.

4. Écrire la même fonction `distance_np(im1:array, im2:array)->float` mais en utilisant la fonction `np.linalg.norm()` de la librairie numpy.

```
Tester : caract_test=lire_donnees_ref("base_référence",
                                      ["Zurich Light BT_majuscules18_10.png"])
         >>> distance(caract_ref['K'][0],caract_ref['K'][0])
         0.0
         >>> distance(caract_test['K'][0],caract_ref['K'][0])
         8.355455053460023
         >>> distance(caract_test['K'][0],caract_ref['K'][1])
         8.961496430503434
         >>> distance(caract_test['K'][0],caract_ref['K'][2])
         9.905886863045481
         >>> distance_np(caract_ref['K'][0],caract_ref['K'][0])
         np.float32(0.0)
         >>> distance_np(caract_test['K'][0],caract_ref['K'][0])
         np.float32(8.355455)
         >>> distance_np(caract_test['K'][0],caract_ref['K'][1])
         np.float32(8.961497)
         >>> distance_np(caract_test['K'][0],caract_ref['K'][2])
         np.float32(9.905888)
```

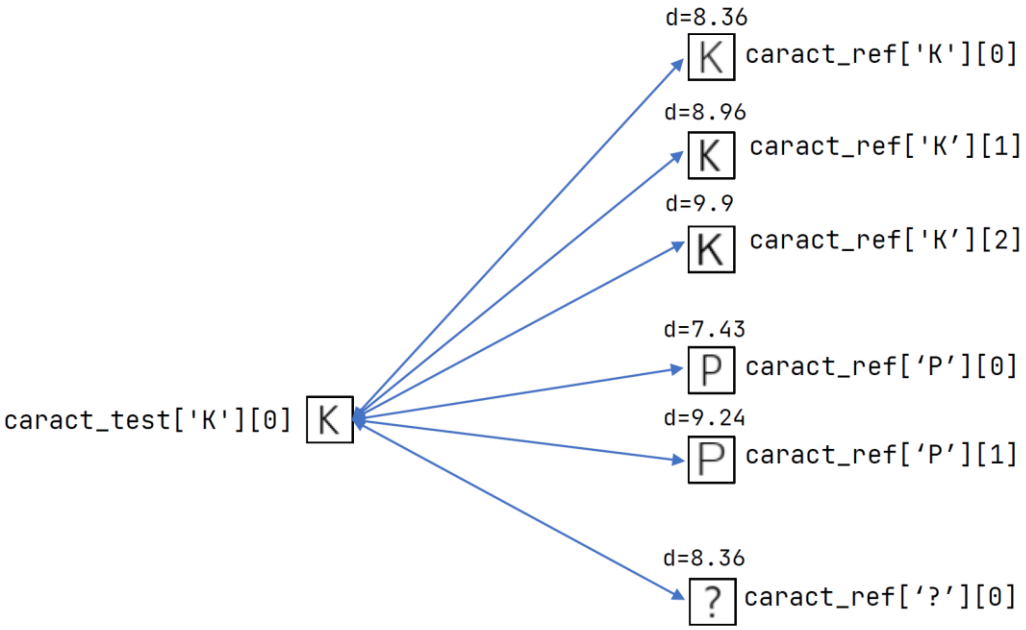
Pour information, les distances calculées dans les exemples précédents sont les suivantes :



5. Écrire la fonction `calcul_distances(carac_ref:dict, carac_test:array)->dict` qui prend en argument le dictionnaire des tableaux catégorisés et un tableau associé au caractère à tester et qui renvoie le dictionnaire des distances.

```
Tester : >>> calcul_distances(caract_ref,caract_test['K'][0])
 {'K': [np.float32(8.355455), np.float32(8.961497),np.float32(9.905888)],
  'P': [np.float32(7.4346333), np.float32(9.240917)],
  '?': [np.float32(8.358953)]}
```

Pour information, les distances calculées sont les suivantes :



La suite consiste à déterminer les K plus petites distances et à extraire les clés correspondantes, puis parmi ces clés déterminer la clé majoritaire. Une méthode envisageable est de trier les distances par ordre croissant pour prendre les K premiers éléments. On suppose qu'il y a au total n images de caractères de référence sur l'ensemble des symboles.

6. En se plaçant dans le pire des cas, indiquer le nom d'une méthode de tri performante envisageable, en précisant sa complexité temporelle en fonction de n.

Une méthode plus efficace est envisagée pour extraire directement les K plus petits éléments. Elle consiste à construire par tri par insertion la liste de taille K. L'algorithme correspondant est donné ci-dessous.

7. Compléter les 3 zones manquantes dans cet algorithme.

```
def Kvoisins(distances:dict,K:int)->list:
    voisins = [(float("inf"),"") for k in range(K)]
    for lettre in distances:
        d = distances[lettre]
        for j in range( ..............  ):

            if ..........................  :
                k = len(voisins)-1

                while ..............................  :
                    voisins[k] = voisins[k-1]
                    k = k - 1
                voisins[k] = [d[j], lettre]
    return voisins
```

```
Tester : >>> Kvoisins(distances,6)
         [[np.float32(7.4346333), 'P'], [np.float32(8.355455), 'K'],
         [np.float32(8.358953), '?'], [np.float32(8.961497), 'K'],
         [np.float32(9.240917), 'P'], [np.float32(9.905888), 'K']]
```

8. Préciser la complexité temporelle asymptotique dans le pire des cas de cet algorithme en fonction de `n` et de `K`. Comparer avec l'utilisation d'un tri classique sachant que `n` est grand et `K` ne dépassera pas 5.

9. Écrire une fonction `symbole_majoritaire(voisins:list)->str` qui, à partir de la liste voisins renvoyée par la fonction `Kvoisins` renvoie le symbole majoritaire.

```
Tester :  >>> voisins = Kvoisins(distances,5)
          >>> symbole_majoritaire(voisins)
          'P'
```

On teste l'algorithme sur les caractères extraits dans la partie précédente ("Beauté,"). On obtient les résultats suivants :

Nombre de voisins K	Type d'éléments dans la base de données	Nombre d'éléments dans la base n	Caractères obtenus
1	fonte similaire au texte analysé	79 images correspondant aux 79 symboles	`"Bssi!-,"`
4	fonte similaire au texte analysé	79 images correspondant aux 79 symboles	`"Bssi!-,"`
1	40 fontes proches de celle du texte analysé	40*79 images correspondant aux 79 symboles	`"Bsauté,"`
4	40 fontes proches de celle du texte analysé	40*79 images correspondant aux 79 symboles	`"Bsauté,"`
1	40 fontes pour 8 polices différentes	320*79 images correspondant aux 79 symboles	`"Beauté,"`
4	40 fontes pour 8 polices différentes	320*79 images correspondant aux 79 symboles	`"Beauté,"`

10. Commenter les résultats obtenus.

Le sujet s'arrête sur cette dernière question, mais nous allons continuer pour mettre en œuvre l'algorithme KNN sur quelques exemples et tester sa robustesse.

IV) EXPÉRIMENTATIONS SUR LA RECONNAISSANCE DE MOTS

Le dossier « test_mot » contient les symboles de la phrase "Beauté," après numérisation et traitement.

11. Compléter la fonction `Lire_test_mot(repertoire)` qui renvoie un dictionnaire sous la forme :

```
symboles_numpy = {0 : (array_image_numpy_image00, symbole_image00),
                  1 : (array_image_numpy_image01, symbole_image01),
                  ...}
```

Où chaque array correspond au contenu numérique des images dans le répertoire « test_mot ».

Par exemple pour le mot "Beauté," :

```
{0: (array([[...]], 'B'), 1: (array([[...]], 'e'), ...}
```

La fonction `liste_fichiers_repertoire(repertoire)` permet de retourner la liste de tous les fichiers contenus dans un répertoire.

Nous allons maintenant utiliser les bases d'apprentissage « base_référence » et « base_11x79 » afin de reconnaître le mot avec la méthode KNN.

12. Écrire la fonction `KNN_test(symboles_numpy, k, base)` qui retrouve le mot codé dans le dictionnaire `symboles_numpy` :
- Récupère la liste de tous les fichiers contenus dans le dossier `base` ;
- Construit le dictionnaire `carac_ref` contenant l'ensemble des array des images du dossier `base` ;
- Pour chaque symbole dans le dictionnaire `symboles_numpy` :
    - Calcule les distances entre ce symbole et les images dans le dictionnaire `carac_ref` ;
    - Récupère les k plus proches voisins ;
    - Sélectionne le meilleur candidat par vote majoritaire

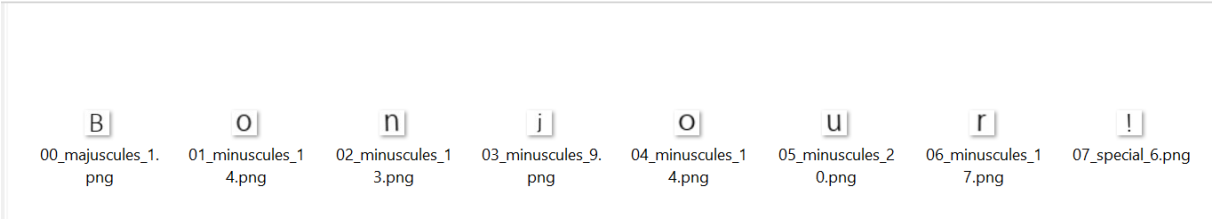
Tester :

```
>>> KNN_test(symboles_numpy,1,"base_référence")
["'", "'", 'a', 'Q', 'l', "'", 'r']
>>> KNN_test(symboles_numpy,1,"base_11x79")
['B', 'e', 'a', 'u', 't', 'é', ',']
```

Si vous souhaitez faire des essais sur d'autres mots, vous pouvez construire un mot avec le script « construction_test_mot.py » en lançant la commande :

- python construction_test_mot.py --mot "Bonjour!"

Cela créera les images « numérisées » du mot dans le répertoire « test_mot » :




```
>>> KNN_test(symboles_numpy,1,"base_référence")
["'", "'", 'q', 'j', "'", 'q', "'", '(']
>>> KNN_test(symboles_numpy,1,"base_11x79")
['B', 'o', 'n', 'j', 'o', 'u', 'r', 'I']
>>> KNN_test(symboles_numpy,4,"base_11x79")
['B', 'o', 'n', 'j', 'o', 'u', 'r', '!']
```

Si votre PC le permet, vous pouvez également essayer avec la « base_41x79 » :

```
>>> KNN_test(symboles_numpy,1,"base_41x79")
['B', 'o', 'n', 'j', 'o', 'u', 'r', '!']
```

V) MATRICE DE CONFUSION

Les tests précédents ont mis en évidence que notre algorithme KNN n'est pas parfait et se trompe quelques fois sur l'identification des symboles (en particulier lorsque la base d'apprentissage est petite).

Pour évaluer le succès de l'algorithme KNN, on utilise une matrice de confusion. Dans notre application, nous avons deux types de données :

- Les symboles appris ;
- Les symboles recherchés.

La matrice est définie ainsi :

- Chaque ligne correspond à un symbole recherché ;
- Chaque colonne représente le symbole trouvé.

En nous adaptant aux données de ce TP, cela donne :

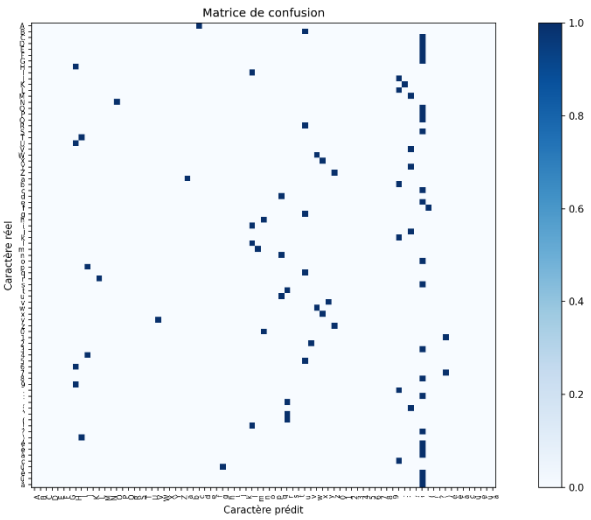
- Ligne l : index du symbole dans la liste `symboles_confusion=["ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789,.:,;'(!?)éèàçùêûâ"]`
- Colonne c : index du symbole trouvé dans la même liste.

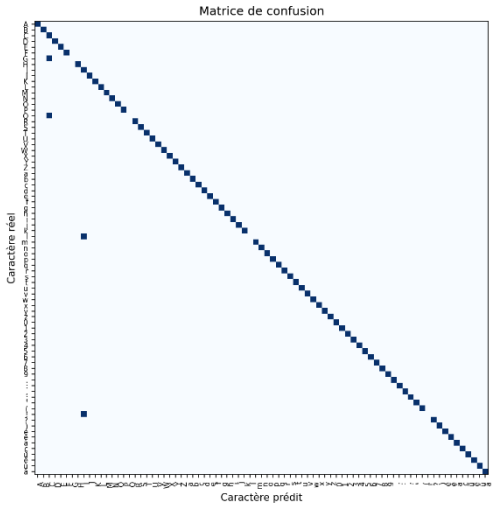
13. Indiquer ce que voudrait dire M(1,0) = 1.

Les 79 symboles « numérisés » à utiliser pour construire la matrice de confusion sont dans le dossier « test_confusion ».

14. Écrire la fonction `Matrice_confusion(k,base)` qui retourne la matrice de confusion sur les symboles de la liste `symboles_confusion` pour un KNN avec les k plus proches voisins et en utilisant la base `base` comme source d'apprentissage.

```
Tester :    >>> matrice = Matrice_confusion(1,"base_référence")
            >>> afficher_matrice_confusion(matrice)
            >>> matrice = Matrice_confusion(1,"base_11x79")
            >>> afficher_matrice_confusion(matrice)
```





15. Visuellement, que pouvez-vous conclure du KNN à partir de ces matrices de confusion ? Tester également avec d'autres valeurs de K.

Le taux de réussite est calculé avec la relation suivante :

$$\text{Acc} = \frac{tr(M)}{\sum_{i,j} M_{i,j}}$$

16. Écrire la fonction `Taux_de_reussite(matrice)` qui calcule le taux de réussite.

Tester :

```
>>> matrice = Matrice_confusion(1,"base_référence")
>>> Taux_de_reussite(matrice)
0.10126582278481013
>>> matrice = Matrice_confusion(1,"base_41x79")
>>> Taux_de_reussite(matrice)
0.9493670886075949
>>> matrice = Matrice_confusion(4,"base_11x79")
>>> Taux_de_reussite(matrice)
0.9620253164556962
```