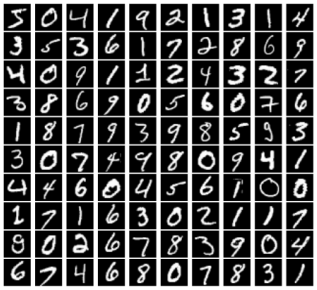
TP : Apprentissage

1 Reconnaissance de chiffres via l'algorithme des plus proches voisins



Mise en place. Sur la page web, télécharger l'archive `apprentissage.zip`. Dézipper là (Clic droit, extraire, ou quelque chose dans ce goût là). Télécharger également le fichier `annexe_TP_apprentissage.py` et ouvrez le avec Pyzo. Modifier le chemin en haut dans le `os.chdir` pour qu'il pointe vers le répertoire `apprentissage` créé. Exécutez `test_debut()` :

- si le message vous dit que vous pouvez commencer le TP, commencez le TP ;
- si le message vous dit qu'il y a une erreur et de m'appeler, appelez-moi ;
- si rien ne se passe, vous n'avez pas d'appelé `test_debut()` ou oublié de mettre les parenthèses.

Question 1. *Récupération des données.* Dans le dossier `apprentissage` (votre répertoire de travail courant, donc), 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). La première étape est de construire une liste de la forme `(image, chiffre représenté)`. Pour ça il va falloir :

- récupérer la liste des noms d'images dans le répertoire courant. Pour ça, rien de plus simple : il suffit d'utiliser `os.listdir()` (tester dans la console), qui vous donne la liste des fichiers présents dans le répertoire courant. Ne garder que ceux qui terminent pas `png`.
- ouvrir chacune des images : pour chaque nom `s` dans la liste précédente, `Image.open(s)` renvoie l'image ouverte en Python ;
- récupérer de chaque nom d'image le dernier chiffre `y` (qu'il faudra convertir en entier).

1. Écrire une fonction `image_assoc(s)` prenant en entrée un nom d'image comme `'Image_000_5.png'`, et renvoyant le tableau Numpy associé à l'image.

```
>>> image_assoc('Image_000_5.png').show() #l'image s'affiche !
```

2. Écrire une fonction `chiffre_assoc(s)` prenant en entrée un nom d'image comme `'Image_000_5.png'`, et renvoyant le dernier entier (sous la forme d'un entier !).

```
>>> chiffre_assoc('Image_000_5.png')
5
```

3. En déduire une fonction `recupere_images()`, sans argument, renvoyant une liste de couples `(image, chiffre)`, de taille 200, contenant les images et les chiffres représentés (rappel : `os.listdir()` et ne garder que les noms de fichiers d'extension `png`).

```
>>> len(recupere_images())
200
```

Remarques : j'ai trié la sortie de `os.listdir()`, si vous faites de même, on devrait avoir les mêmes résultats dans la suite !

Dans la suite, faites l'appel `Z=recupere_images()`.

Rappel sur les tableaux Numpy et les images en noir et blanc. Une image en noir et blanc est représenté comme un tableau d'octets, chaque pixel étant représenté par un entier entre 0 et 255 indiquant le niveau de gris (0 signifie noir, 255 signifie blanc).

— Pour `im` une telle image, `np.array(im)` permet de récupérer le tableau associé :

```
>>> np.array(Z[0][0]) #Z[0] : premier couple, Z[0][0] : première image.
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
...] # je n'ai pas tout affiché !
```

— les dimensions d'un tel tableau (hauteur, largeur), se récupèrent via l'attribut `shape` :

```
>>> np.array(Z[0][0]).shape
(28, 28)
```

— avec `t` un tel tableau, `i` l'indice d'une ligne et `j` l'indice d'une colonne, le pixel en case (i, j) se récupère comme `t[i][j]` ou `t[i,j]` :

```
>>> np.array(Z[0][0])[15,18]
150
```

Attention, le type des pixels est `np.uint8` (entier non signé sur 8 bits, c'est-à-dire entre 0 et 255). Attention dans la suite, ce type réserve quelques surprises lorsqu'on fait des opérations et que le résultat sort de l'intervalle de représentation (le résultat est modulo 256 sur les entiers).

```
>>> np.uint8(150)-np.uint8(186)
<stdin>:1: RuntimeWarning: overflow encountered in ubyte_scalars
220
```

Dans la suite, on pourra récupérer un tableau flottant via `np.array(objet, dtype=float)`.

```
>>> np.array(Z[0][0], dtype=float) #le meme tableau, mais flottant !
```

Distances entre deux images. Dans la suite, on utilisera la distance suivante pour deux images en noir et blanc de même taille (h, ℓ) :

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

Avec $I_{i,j}$ la valeur du pixel en case (i, j), de même pour $I'_{i,j}$.

Question 2. Écrire une fonction `distance_images(I,I2)` renvoyant la distance entre les deux images, supposées de même taille (utiliser des tableaux flottants associés aux images).

```
>>> distance(Z[0][0], Z[1][0])
29506.0
```

Algorithme des k-plus proches voisins. On rappelle l'algorithme des k-plus proches voisins :

Algorithme 1 : Algorithme des k plus proches voisins

Entrée : Un ensemble fini de points X d'un espace normé (E, d), un entier $k \leq |X|$. Pour chaque $x \in X$, on connaît sa classe $c(x)$. Un point $y \in E$.
Sortie : Une valeur $c(y)$
Chercher les k plus proches éléments $x \in X$ de y pour la distance d : $x_1, \ldots, x_k$;
Renvoyer la classe majoritaire parmi les $(c(x_i))_{1 \leq i \leq k}$;

Dans la suite, on va utiliser `Z[:150]` comme ensemble d'apprentissage et `Z[150:]` comme ensemble de test. On prendra donc `X=Z[:150]` comme ensemble dans l'algorithme en pseudo-code et `Y=Z[150:]` comme ensemble de test.

Question 3. *Implementation de l'algorithme.* Écrire une fonction `k_voisins(X, k, im)` prenant en entrée :

— une liste `X` constitué de couples `(image, classe)`, où la classe est ici un entier de $[\![0, 9]\!]$;

— un entier $k < |X|$;

— une image `im`.

La fonction renverra la classe majoritaire parmi les `k` plus proches voisins de `im` dans `X`.

```
>>> k_voisins(X,10,Z[151][0]) #reponse correcte
6
>>> k_voisins(X,10,Z[150][0]) #reponse erronee, c'est un 4 !
1
```

Matrice de confusion. Pour décider quel est le k optimal dans notre algorithme, on va utiliser l'ensemble de test `Y`, qui est un autre ensemble d'images dont on connaît les chiffres associés. La matrice de confusion est définie comme suit : $C = (n_{i,j})_{0\leq i,j<c}$ où $[\![0, c-1]\!]$ est l'ensemble des classes possibles ($c = 10$ pour nous) et $n_{i,j}$ est le nombre d'éléments de l'ensemble de test de la classe i attribués à la classe j par l'algorithme avec X. Cette matrice dépend de l'entier k choisi.

Question 4. Écrire une fonction `mat_confusion(X,Y,k)` renvoyant la matrice de confusion de taille 10×10.

```
>>> mat_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.]])
```

Rappel : `np.array((10,10))` permet de créer une matrice de taille 10×10 remplie de zéros.

Question 5. Écrire une fonction `dist_diag(M)` prenant en entrée une telle matrice carrée, et faisant la somme des éléments hors-diagonale : si `M` est une matrice de confusion, c'est le nombre d'éléments de l'ensemble de tests Y incorrectement classés.

Question 6. Pour optimiser l'approche, il reste à trouver l'entier $k \in [\![1, |X|-1]\!]$ qui minimise cette quantité. Combien trouvez-vous ?

2 Algorithme des k-moyennes

Algorithme 2 : Algorithme des k-moyennes

Entrée : Un ensemble fini de points X d'un espace euclidien, un entier $k \leq |X|$
Sortie : Une partition de X en parties non vides $X_0, \dots, X_{k-1}$
Répartir les points de X en sous-ensembles disjoints non vides $X_0, \dots, X_{k-1}$;
tant que *la situation évolue* **faire**
  Calculer $m_0, \dots, m_{k-1}$, barycentres de $X_0, \dots, X_{k-1}$;
  pour *chaque point x de X* **faire**
    Trouver i tel que $\|x - m_i\|$ est minimal parmi les $\|x - m_j\|$, et placer x dans X_i
Renvoyer $X_0, \dots, X_{k-1}$

Le but est simplement de recoder l'algorithme des k-moyennes (ne regardez pas le cours !)

2.1 Fonctions utiles

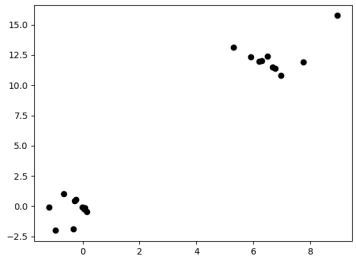
Dans la suite, les fonctions prennent en entrée des tableaux Numpy de même taille (disons p) représentant des vecteurs ou des points de $\mathbb{R}^p$ muni de son produit scalaire canonique.

Question 7. Recoder rapidement les fonctions suivantes :

- `ps(x,y)` calculant le produit scalaire de deux vecteurs représentés par des tableaux Numpy ;
- `dist(x,y)` calculant la distance entre deux points représentés par des tableaux Numpy ;
- `plus_proche(p,M)` : avec `p` un point et `M` une liste de points, renvoie *l'indice* dans `M` du point le plus proche de `M`.
- `barycentre(X,xi)` : avec `X` un ensemble de points et `xi` une liste non vide d'indices de `X`, renvoie l'isobarycentre des points $\{$`X[i]` $\mid i \in$ `xi`$\}$.

2.2 Code de la fonction

Question 8. *k-moyennes*. Dans l'annexe se trouve la fonction `initialisation(n,k)` renvoyant une partition de $[\![0, n-1]\!]$ en k sous-ensembles non vides, sous réserve que $k \leq n$. Implémenter l'algorithme des k-moyennes sous la forme d'une fonction `k_moyennes(X,k)`. Votre fonction renverra une partition de l'ensemble X sous la forme d'une liste de listes d'indices.



Les fichiers `nuage1.txt` (représenté ci-dessus), `nuage2.txt` et `nuage3.txt` contiennent des points du plan, sous la forme de lignes `x;y` où `x` et `y` représentent des nombres flottants. On rappelle que :

- `f=open(nom_fichier,'r')` permet d'ouvrir le fichier dont le nom est la chaîne de caractères `nom_fichier` en lecture ;
- `for ligne in f` : permet de faire parcourir à la variable `ligne` les lignes du fichier (le saut de ligne `'\n'` est présent en chaque fin de ligne) ;
- si `s` est une chaîne de caractères et `c` un caractère, `s.split(c)` s'évalue en une liste obtenue en séparant `s` autour de `c`. Par exemple, `L="a;bb;ccc".split(";")` affecte à `L` la liste `["a", "bb", "ccc"]`.
- `float(x)` s'évalue en la conversion de `x` en flottant. Par exemple, `float("3.4")` est de flottant `3.4`.

Question 9. Lire les trois fichiers et stocker les ensembles de points obtenus comme liste de tableaux Numpy dans les variables `X1`, `X2`, `X3`. On pourra par exemple pécrire une fonction `lire_fichier(s)` prenant en entrée la chaîne de caractères du nom du fichier.

Question 10. Tester sur ces ensembles votre fonction `k_moyennes(X,k)`. *Remarque :* parfois, une des listes de la partition se retrouve vide, ce qui produit une erreur dans le calcul des barycentres. Appeler la fonction `ajustement` sur votre liste de listes d'indices encodant une partition (elle modifie la liste et ne renvoie rien) pour éviter ce cas.

2.3 Choisir k : la méthode du coude (elbow)

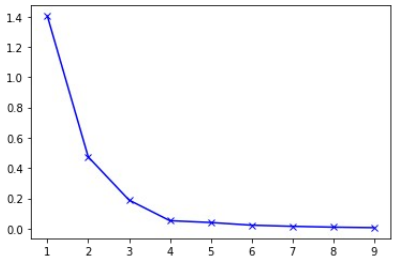
Question 11. Écrire une fonction `somme_dist2(X,par)` prenant en entrée une liste de taille n de points du plan (donné sous la forme de tableaux Numpy), et `par` une partition de `X` en sous ensembles non vides donnés par les listes d'indices, et renvoyant la valeur

$$f(\texttt{X}, \texttt{par}) = \sum_{p \in \texttt{par}} \sum_{i \in p} (\texttt{X}[i] - m_i)^2$$

où m_i est le barycentre de la partie $\{\texttt{X[i]} \mid i \in p\}$.

Question 12. Pour chaque fichier, tracer en abscisse k (variant entre 1 et 10 par exemple), et en ordonnée la quantité précédente, obtenue après application de l'algorithme des k-moyennes avec comme paramètre k.

On obtient naturellement une courbe décroissante : plus k est grand, plus $f(\texttt{X}, \texttt{par})$ est petit. L'allure du graphique est semblable à l'allure d'un coude comme ci-dessous :



Comme on le voit, f diminue très vite, pour ensuite ne plus trop évoluer. Le k optimal se trouve au niveau du « coude ». Sur le graphique ci-dessus, le coude se situe entre $k = 2$ et $k = 4$.

Question 13. Repérer le k optimal pour chacun des fichiers. Tracer les points sur un graphique (si `x` est la liste des abscisses et `y` la liste des ordonnées des points du nuage, `plt.plot(x,y,'o')` trace les points). Est-ce cohérent ?