TP D'INFORMATIQUE N°5
Algorithme des k plus proches voisins

1 Implémentation de knn

Dans la suite, on identifiera un point au tuple de ses coordonnées, et on appelera *point annoté* un couple formé d'un point et d'une étiquette (entière).

1. Écrire une fonction `distance` prenant en argument deux points ayant le même nombre de coordonnées, et renvoyant la distance euclidienne entre ces points.
2. On veut à présent écrire une fonction `liste_voisins` prenant en argument un point `p`, un entier `k`, et un échantillon d'entraînement (ie une liste de points annotés) `Ltrain`, et renvoyant la liste des k points annotés les plus proches du point p.

   Implémenter une version de cette fonction utilisant la fonction `sorted` de Python : `sorted(L, key = f)` renvoie une version de la liste `L` triée par valeurs croissantes des `f(x)`, `x` $\in$ `L`. On prendra bien soin de définir préalablement la bonne fonction `f` à l'intérieur de la définition de `liste_voisins`. On notera bien que cette fonction `f` doit prendre en argument un élément de `L`, c'est-à-dire un point annoté.
3. Écrire une fonction `etiquette_maj` prenant en argument une liste de points annotés `V`, et renvoyant une étiquette majoritaire dans `V`, c'est-à-dire dont le nombre d'occurrences est maximal.
4. Écrire une fonction `knn` prenant en argument un point `p`, un entier `k` et un échantillon d'entrainement `Ltrain`, et renvoyant l'étiquette prédite par l'algorithme des k plus proches voisins pour le point `p`.

2 Test sur la reconnaissance de chiffre

Pour tester la fonction `knn`, nous allons importer une banque d'exemples annotés d'images de chiffres disponible dans la bibliothèque `scikit-learn`.

1. Importer cette banque en recopiant le code suivant :

```
from sklearn.datasets import load_digits

digits = load_digits()
X = digits.data
Y = digits.target
```

   `X` contient alors la liste des 1797 points de la banque. Chaque point correspond à une image de 8×8 pixels, représentée par une liste de 64 nombres. `Y` correspond aux 1797 annotations associées à ces points, ainsi `Y[0]` indique le chiffre représenté sur l'image codée dans `X[0]`. On peut afficher cette image avec le code suivant :

```
import matplotlib.pyplot as plt

plt.imshow(digits.images[0], cmap = 'gray')
plt.show()
```

2. Combiner les deux listes X et Y en une seule liste L de points annotés.
3. Importer la bibliothèque `random`, et utiliser la fonction `shuffle` de cette bibilothèque pour mélanger `L`. Former ensuite une liste `Ltrain` formée des 80% premiers éléments de `L`, et `Ltest` formée des éléments restants.
4. Utiliser `Ltrain` et `Ltest` pour tester votre fonction `knn` avec différentes valeurs de k.

5. Pour chaque valeur de k de 1 à 15, afficher le taux d'erreur empirique sur `Ltest` de `knn` exploitant `Ltrain`. Proposer une valeur de k permettant de minimiser ce taux d'erreur.
6. Compléter l'affichage précédent pour afficher également la matrice de confusion dans chaque cas.
7. Déterminer une valeur de k permettant de minimiser la somme pour chaque point de `L2test` de $|e_c - e_p|$, où e_c est l'étiquette annotée et e_p l'étiquette prédite par `knn`.

3 Test sur une nouvelle image

On se propose maintenant de créer nous-même l'image sur laquelle nous allons tester notre version de `knn`.

1. Créer avec paint une image de dimensions 8×8 représentant un magnifique chiffre. Enregistrer cette image au format png dans le répertoire de travail de Pyzo.
2. Charger cette image dans Python à l'aide de la fonction `imread` de la bibliothèque `matplotlib.pyplot`, prenant en entrée le nom du fichier.
3. Afficher l'image avec les fonctions `imshow` et `show`.
4. Il faut à présent passer l'image au même format que les points manipulés par `knn`. Observer le format actuel de l'image. Par exemple, chaque pixel peut correspondre à une liste de quatre valeurs entre 0 et 1. Il faut alors :
   - créer une nouvelle image de la bonne taille ;
   - Définir chaque pixel de cette nouvelle image comme la première valeur du pixel correspondant dans l'image de départ (passage en noir et blanc) ;
   - remplacer chaque valeur de pixel x par $1 - x$ (pour obtenir un chiffre tracé en blanc sur fond noir) ;
   - multiplier chaque valeur de pixel par 16 (pour utiliser la même plage de valeurs que dans la banque utilisée) ;
   - aplatir l'image 8×8 en une liste de longueur 64, en concaténant chacune des 8 lignes de l'image.
5. Tester `knn` sur le point obtenu à partir de votre image, avec une valeur de k judicieusement choisie.

4 Variantes

1. On se propose d'écrire une autre implémentation de la fonction `Liste_voisins`, maintenant à jour une liste des k points les plus proches de p rencontrés jusque-là dans `Ltrain`.
   (a) Écrire une procédure `inserer` prenant en argument un point `p`, un point annoté `p2` et une liste de points annotés `V` supposée triée selon la distance à `p`, et insérant `p2` dans `V` de façon à la maintenir triée.
   Par exemple, si `p = (0, 1)` et `V = [((0, 0.8), 0), ((0, 1.3), 1), ((0, 2), 0)]`, à l'issue de l'appel `inserer(p, ((0.4, 1), 1), V)`, on aura comme nouvelle valeur `V = [((0, 0.8), 0), ((0, 1.3), 1), ((0.4, 1), 1), ((0, 2), 0)]`.
   (b) Implémenter une nouvelle version de `liste_voisins` partant d'une liste vide et y insérant chaque élémennt de `Ltrain`. Une fois que la liste produite aura atteint une longueur de `k`, on enlevera après chaque insertion son élément maximal avec la fonction `pop()`.
2. Déterminer la complexité des deux implémentations de `Liste_voisins`, en fonction de k et de la longueur n de `Ltrain`. On supposera que la fonction `distance` est en $O(1)$, et que la fonction `sorted` est en $O(n \log n)$.
3. Concevoir et implémenter une version de `knn` attribuant à un voison de p d'autant plus de votes qu'il est proche de p.