IA - Apprentissage supervisé - Classification
Algorithme des k plus proches voisins kNN

IA - Apprentissage supervisé - Classification

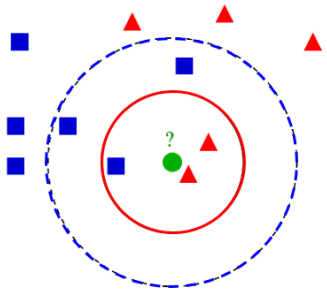
📚 En intelligence artificielle, plus précisément en **apprentissage automatique**, la méthode des k plus proches voisins est une méthode d'**apprentissage supervisé**.
En abrégé k-NN ou KNN, de l'américain k-nearest neighbors.

📚 Entrées et sorties de l'algorithme

On dispose d'un **ensemble E** d'objets o de même structure et d'un **sous-ensemble d'apprentissage A** constitué N objets o_i de E représentant des **données classées ou étiquetées** (données représentées par des carrés bleus ou des triangles rouges ci-dessous).

Les données étiquetées sont réparties en **classes**, chaque classe étant associée à une étiquette (la couleur et la forme dans cet exemple).
Ces étiquettes sont choisies par un **expert humain**, c'est en ce sens qu'on parle d'**apprentissage supervisé**.
Ces classes forment une partition de l'ensemble d'apprentissage A (sous-ensembles de A non vides et disjoints deux à deux dont la réunion est A).

L'objectif de la méthode est d'affecter une étiquette à un objet o (cercle vert) de E qui n'est pas dans l'ensemble d'apprentissage A (i.e. il n'est pas étiqueté) en déterminant l'étiquette majoritaire parmi les étiquettes de ses plus proches voisins.



Si $k = 3$ (cercle rouge en ligne pleine), l'objet sans étiquette (cercle vert) est affecté à la classe des triangles rouges (2 triangles / 1 carré à l'intérieur du cercle rouge définissant les 3 plus proches voisins).
Si $k = 5$ (cercle en pointillés bleus), l'objet est affecté à la classe des carrés bleus (3 carrés / 2 triangles à l'intérieur du cercle bleu).

En dehors des données elles-mêmes, l'algorithme admet donc deux paramètres un **entier** $k < N$ et une **distance** (ou mesure de similarité) $d(o_i, o_j)$ entre deux objets, à définir (dans l'exemple, il s'agit de la distance Euclidienne dans le plan).
Suivant le problème envisagé, différentes définitions peuvent être utilisées pour définir une telle distance.

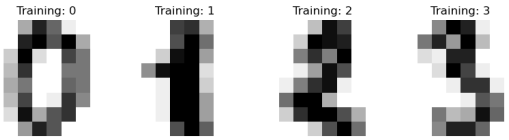
Il s'agit donc de trouver, parmi les objets de l'ensemble d'apprentissage A, les k plus proches de l'objet o sur le critère de la distance d et de déterminer l'étiquette majoritaire parmi ceux-ci : on affecte ainsi l'objet à l'une des classes, on parle d'algorithme de **classification**.

Objectif ✔️ : classer un objet parmi d'autres objets répartis en différentes catégories à partir de ses caractéristiques.

Savoir faire✏️ : écrire une bibliothèque de fonctions et utiliser bibliothèque sklearn (https://scikit-learn.org/stable/index.html).

Exemple - OCR (optical character recognition)

💡 Objectif : à partir de l'image noir et blanc, 8x8 pixels, d'un chiffre manuscrit c (compris entre 0 et 9) scanné, on souhaite identifier c.
Ce chiffre représente la classe de l'objet, il y a donc 10 classes de (0 à 9).
Exemple d'images issues de l'ensemble d'apprentissage et leurs étiquettes (chiffre au-dessus de l'image) :



L'objet o à classer est ici une image définie par un tableau numpy de 8 lignes et 8 colonnes contenant des valeurs représentant des niveaux de gris.

```
o = np.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.] ])
```

La commande o.flatten() permet de convertir cet objet en tableau ou vecteur de dimension $n = 8 \times 8 = 64$ coordonnées :

```
array([ 0.,  0.,  5.,  13.,  9.,  1.,  0.,  0.,  ...  ,  0.,  0.,  6.,  13.,  10.,  0.,  0.,  0.])
```

Rq : La commande o.reshape(8,8) permet de revenir au tableau 8×8.

💡 **Conclusion**
L'algorithme k-NN est capable de traiter des objets de type quelconque dès que ceux-ci sont ramenés à des **vecteurs de dimension** n (n est appelé dimensionality dans sklearn, cf. dernière partie du TP).

Cet exemple sera abordé en fin de TP.

1 Ecriture d'une bibliothèque de fonctions

Notations - Définitions

- **Objet** o : tableau de dimension n caractérisant l'objet.
- **E** : ensemble d'objets ayant la structure de l'objet o.
- **A** : ensemble d'apprentissage constitué d'objets o_i de E et de leurs étiquettes e_i. A est construit à partir de deux tableaux **data** (objet) et **target** (étiquette) (dénominations utilisées par la bibliothèque sklearn pour désigner les objets et leurs étiquettes), sous la forme d'une liste de tuples de la forme $[(o_0, e_0), (o_1, e_1), ...]$ où e_i est l'étiquette de l'objet o_i.
- **Classes** des objets : ensemble des étiquettes.

Principe de l'algorithme

Entrées :
- un objet o d'un ensemble E ;
- un ensemble d'apprentissage A d'objets étiquetés représenté par une liste de la forme $[(o_0, e_0), (o_1, e_1), ...]$;
- un entier strictement positif k qui est le nombre de voisins à considérer ;
- une fonction distance d définie pour les objets de E (cette fonction peut posséder différentes définitions, ce sera donc un paramètre des fonctions de la bibliothèque).

Sortie : une étiquette e pour l'élément o.

📚 **Algorithme**
1. Déterminer les k éléments de l'ensemble d'apprentissage A les plus proches de l'objet o au sens de la distance d.
2. Déterminer l'étiquette la plus fréquente parmi les étiquettes de ces k éléments (choix aléatoire en cas d'égalité).

Remarque : l'implémentation des fonctions ci-dessous ainsi que leurs paramètres sont choisis de façon à être compatibles avec la bibliothèque scikit learn.

Bibliothèques

💻 Exécuter le code.

[ ]:
```
from _validationkNN import *
from math import sqrt
```

Étape 1 - Calcul des distances entre deux objets

Étant donnés deux objets o_1 et o_2 représentés par des vecteurs dans un espace à n dimensions de coordonnées respectivement (x_i) et (y_i), la distance euclidienne s'écrit : $d(o_1, o_2) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$

✏️ Écrire une fonction d_Euclide(o1, o2) renvoyant la distance euclidienne entre les objets o1 et o2.
Rappel : ces objets sont des vecteurs de dimension inconnue a priori (mais identique pour les deux objets).
La fonction sum de python n'est pas autorisée.

[ ]:
```
def d_Euclide(o1, o2):
    """
    Paramètres :
        o1, o2 : 2 objets = 2 tableaux de coordonnées
    Renvoie :
        Distance euclidienne entre o1 et o2
    """
```

[ ]:
```
# Tests distances (distances aisément calculables de tête)
for t in [((0,0),(0,1)), ((0,0),(1,0)), ((0,0),(1,1)), ((0,0),(-2,0))]:
    o1, o2 = t
    print(f'Distance entre {o1} et {o2} : {d_Euclide(o1,o2)}')
```

[ ]:
```
test_d_Euclide()
```

Étape 2 - Plus proches voisins d'un objet

Étape 2.1 - Liste des distances de l'objet o à tous ses voisins dans A

✏️ Écrire une fonction distances(o, A, fct_distance) renvoyant une liste constituée de tuples de la forme $((o_i, e_i), fct\_distance(o, o_i))$ pour tous les objets étiquetés (o_i, e_i) de A.
Rappel : A est une liste de la forme $[(o_0, e_0), (o_1, e_1), ...]$.

[ ]:
```
def distances(o, A, fct_distance):
    """
    Paramètres :
        o :            objet (tableau de coordonnées)
        A :            liste d'apprentissage A de la forme [(o_0, e_0), (o_1, e_1),...]
        fct_distance : fonction utilisée pour calculer la distance
    Renvoie :
        Liste [((o_i, e_i), fct_distance(o_i, o))...] où (o_i, e_i) est un élément de A
    """
```

[ ]:
```
# Test distances
A = [((-4,2),'c'), ((0,0), 'a'), ((1,0), 'b'), ((0,1), 'b'), ((3,-1), 'a')]
distances((-1,0), A, d_Euclide)
```

[ ]:
```
test_distances()
```

Étape 2.2 - Tri des distances de l'objet o à ses voisins

✏️ *sort et sorted*

liste.sort(key=None, reverse=False)
liste.sort() est une méthode de l'objet liste qui effectue un tri en place (*liste* est modifiée).

sorted(*liste*, key=None, reverse=False)
sorted(*liste*) est une fonction définie sur les listes qui ne modifie pas *liste* (renvoie une nouvelle liste).
Le paramètre key parmet de définir un critère de tri à l'aide d'une fonction lambda lorsque les éléments e de liste sont eux-mêmes des listes ou des tuples : key=lambda e: e[i] où i est l'index sur lequel on souhaite trier les éléments.

✏️ Écrire une fonction tri_distances(o, A, fct_distance) renvoyant la liste générée par la fonction distances(o, A, fct_distance) triée par ordre croissant des distances .

[ ]:
```
def tri_distances(o, A, fct_distance):
    """
    Paramètres :
        o :            objet (tableau de coordonnées)
        A :            liste d'apprentissage A de la forme [(o_0, e_0), (o_1, e_1),...]
        fct_distance : fonction utilisée pour calculer la distance
    Renvoie :
        Liste [((o_i, e_i), fct_distance(o_i, o))...] triée par ordre croissant des distances
    """
    # Création de la liste [((o_i, e_i), fct_distance(o_i, o))...]

    # Fonction renvoyant la distance de o_i à o
```

```
    # Tri de distances sur le critère de la distance

```

[ ]:
```
# Test tri
A = [((-4,2),'c'), ((0,0), 'a'), ((1,0), 'b'), ((0,1), 'b'), ((3,-1), 'a')]
tri_distances((-1,0), A, d_Euclide)
```

[ ]:
```
test_tri_distances()
```

Étape 2.3 - Liste des k plus proches voisins de l'objet o

✏️ Écrire une fonction liste_kppv(o, A, k, fct_distance) renvoyant la liste des k plus proches voisins de l'objet o dans l'ensemble d'apprentissage A.
La liste renvoyée, constituée d'objets étiquetés, est de la forme $[(o_i, e_i), ...]$.

[ ]:
```
def liste_kppv(o, A, k, fct_distance):
    """
    Paramètres :
        o :            objet (tableau de coordonnées)
        A :            liste d'apprentissage A de la forme [(o_0, e_0), (o_1, e_1),...]
        k :            nombre de plus proches voisins à considérer
        fct_distance : fonction utilisée pour calculer la distance
    Renvoie :
        La liste des coordonnées des k plus proches voisins de o dans A
    """
    # Tri des points de E sur le critère de la distance

    # Sélection des k plus proches voisins = coords des k premiers termes de la liste

```

[ ]:
```
# Test liste_kppv
A = [((-4,2),'c'), ((0,0), 'a'), ((1,0), 'b'), ((0,1), 'b'), ((3,-1), 'a')]
o = (-1,0)
print(tri_distances(o, A, d_Euclide)[:3]) # 3 premiers éléments de la liste triée
liste_kppv(o, A, 3, d_Euclide)
```

[ ]:
```
test_liste_kppv()
```

Étape 3 - Étiquette majoritaire parmi celles des plus proches voisins

Étape 3.1 - Dictionnaire des occurrences des étiquettes

✏️ Écrire une fonction occurrences_classes(L) où L est une sous-liste de la liste d'apprentissage A (liste de tuples (objet, etiquette)) et qui renvoie le dictionnaire des occurrences des étiquettes.
La fonction renvoie donc un dictionnaire de la forme : $\{e_0 : n_0, e_1 : n_1, ...\}$ où n_i est le nombre d'occurrences de l'étiquette e_i dans les tuples de la liste L.

[ ]:
```
def occurrences_classes(L):
    """
    Paramètre :
        L : sous-liste de  A de la forme [(o0, e0), (o1, e1)...]
    Renvoie :
        {e0: n0, e1:n1, ...} où n_i est le nbre d'occurrences de  e_i dans L
    """

```

[ ]:
```
# Test classes
L = [((-4,2),'c'), ((0,0), 'a'), ((1,0), 'b'), ((0,1), 'b'), ((3,-1), 'a')]
print(occurrences_classes(L))
```

Étape 3.2 - Classe majoritaire parmi les k plus proches voisins de l'objet o

✏️ Écrire une fonction classe_majoritaire(L) renvoyant l'étiquette de la classe majoritaire dans la liste L (sous-liste de la liste d'apprentissage A).

[ ]:
```
def classe_majoritaire(L):
    """
    Paramètre :
        L : sous-liste de A de la forme [(o0, e0), (o1, e1)...]
    Renvoie :
        étiquette (classe) majoritaire dans L
    """
    # Création du dictionnaire des classes

    # Recherche de l'étiquette majoritaire
```

[ ]:
```
# Test classe_majoritaire
L = [((-4,2),'c'), ((0,0), 'a'), ((1,0), 'b'), ((0,1), 'b'), ((3,-1), 'a')]
print(classe_majoritaire(L))
```

Étape 4 - Classification : algorithme kNN

Estimation de la classe de l'objet o à partir d'un ensemble d'apprentissage A

✏️ *zip*

zip(liste, liste ...)

✏️ Écrire une fonction kNN(o, data, target, k, fct_distance) renvoyant l'étiquette de la classe majoritaire des k plus proches voisins de l'objet o à l'aide d'un ensemble d'apprentissage A = ([(o_0, e_0), (o_1, e_1),...]) construit à partir d'un tableau data = $[o_0, o_1, ...]$ d'objets dont les étiquettes sont fournies dans le tableau target = $[e_0, e_1, ...]$.
La fonction kNN doit donc construire A à partir des deux tableaux data et target.

[ ]:
```
def kNN(o, data, target, k, fct_distance):
    """
    Paramètres :
        o :            objet (tableau de coordonnées)
        data :         tableau des coordonnées des objets de A
        target :       tableau des étiquettes associées à data (classes de A)
        k :            nombre de plus proches voisins à considérer
        fct_distance : fonction utilisée pour calculer la distance
    Renvoie :
        La classe (étiquette e) de o (classe maj pour k plus proches voisins)
    """
    # Création de A à partir des tableaux data et target

    # Liste des k plus proches voisins de l'objet o

    # Classe majoritaire
```

[ ]:
```
# Test classification
data = [(-4,2), (0,0), (1,0), (0,1), (3,-1)]
target = ['c', 'a', 'b', 'b', 'a']
o = (-1,0)
k = 3

print(kNN(o, data, target, k, d_Euclide))
```

[ ]:
```
test_kNN()
```

2 Exemple simple - Visualisation graphique

Dans cette partie, en vue d'obtenir des représentations graphiques aisées à interpréter, on considère que :
- les objets de l'ensemble E sont des vecteurs de dimension 2, ils seront donc représentés par des points dans un plan ;
- chaque point de A possède une étiquette e_i représentée par une couleur c_i.

Si on ne considère que deux couleurs, par exemple rouge et bleu, chaque point est soit rouge soit bleu.
Il y a alors deux classes de points dans A : les points rouges et les points bleus (étiquette c = 'r' ou c = 'b').

[ ]:
```
from random import random, uniform, randint, choice, gauss, shuffle, seed
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, Rectangle
```

Création des données d'apprentissage

La fonction tirage_gauss(n, classes, mu=100, sigma=40) disponible dans l'une des bibliothèques permet de créer n points répartis en N_C couleurs listées dans le paramètre classes.
Ces points sont distribués selon une loi Gaussienne autour de centres tirés aléatoirement (mu et sigma sont respectivement la plus grande valeur de la moyenne et la plus grande valeur de l'écart-type pour le tirage des abscisses et des ordonnées chaque classe).

💻 Exemple.

[ ]:
```
# 10 points répartis dans 2 classes (couleurs rouge et bleu)
data, e = tirage_gauss(10, ['r', 'b'])
print(f"Coordonnées : {data} \nEtiquettes : {e}")
```

Représentation graphique

La fonction nuage(o, data, target, k, fct_distance) ci-dessous permet de représenter les points de l'ensemble d'apprentissage et le point o à classer.

Les paramètres de la fonction data sont :
- o (tuple de 2 coordonnées) = objet à classer ;
- data (liste de tuples de 2 coordonnées) = objets de l'ensemble d'apprentissage A ;
- target (liste de couleurs) = étiquettes associées aux objets de data ;
- k (int) = nombre de plus proches voisins à prendre en considération ;
- fct_distance = nom de la fonction utilisée pour calculer la distance entre 2 objets.

✏️ *module random*

random.seed(a=None)
Initialise le générateur de nombres aléatoires : en fixant la valeur a, le même tirage aléatoire est obtenu sur différentes machines ou au cours d'exécutions successives.

Exemple d'exécution

💻 💡 Exécuter le code et observer.

[ ]:
```
# Initialisation générateur nombres pseudo-aléatoires = > répétabilité
seed(20) # Placer un commentaire ou modifier la valeur pour modifier le tirage

# Ensemble d'apprentissage
n = 80                          # Nombre de points
classes = ['r', 'b', 'y']       # Couleurs des points = classes

# Création des objets (coords des points) et des étiquettes (couleur associée)
data, target = tirage_gauss(n, classes, mu=200)

# Nombre de k plus proches voisins à considérer
k = 5
```

```
# Objet à classer choisi "manuellement"
o = (100,100)

# Graphe
nuage(o, data, target, k, d_Euclide)
```

Expérimentation

🛠️ 👁️ Expérimenter, en particulier en faisant varier le nombre de k voisins : modifier le code précédent en faisant varier k sans changer l'initialisation du générateur aléatoire puis modifier la « graine » (valeur fournie à la fonction seed ligne 2).

Tirage aléatoire du point à classer

✏️ *module random*

random.random()
Renvoie un nombre aléatoire à virgule flottante x tel que $0.0 \leq x < 1.0$

✏️ *python round*

round(x, n)
Arrondit le flottant x avec n chiffres après la virgule.

Coordonnées aléatoires

✏️ Écrire une fonction tirageCoords() renvoyant 2 flottants aléatoires x et y, arrondis à deux chiffres après la virgule, tels que :$0.0 \leq x, y < 100.0$ (cf. rappels ci-dessus).

[ ]:
```
def tirageCoords(maxi=100):
    """
    Paramètres :
        maxi : tirage dans l'intervalle 0 <= x, y < maxi
    Renvoie :
        x, y : tuple de coordonnées (2 chiffres après la virgule)
    """
```

[ ]:
```
# Test tirageCoords
tirageCoords(maxi=100)
```

Représentation graphique

✏️ Tirer un point au hasard et visualiser le résultat de l'algorithme kNN à l'aide de la fonction nuage (cf. "Exemple d'exécution" ci-dessus) avec le même jeu de données étiquetées.

[ ]:
```
# Objet à classer tiré aléatoirement
o = tirageCoords(maxi=200)

# Graphe
nuage(o, data, target, k, d_Euclide, infos='')
```

3 Performances - Matrice de confusion

L'algorithme de k plus proches voisins permet de faire des prédictions, il reste à mesurer la qualité de ces prédictions.

La **matrice de confusion** mesure la qualité d'un système de classification.
Chaque ligne correspond à une classe réelle, chaque colonne correspond à une classe estimée pour un ensemble de données test étiquetées par un expert.
La cellule ligne L, colonne C contient le nombre d'éléments de la classe réelle L qui ont été estimés comme appartenant à la classe C.
Un des intérêts de la matrice de confusion est qu'elle montre rapidement si un système de classification parvient à classifier correctement.

Exemple

On souhaite mesurer la qualité d'un système automatique de classification de courriers électroniques.

Les courriers sont classifiés selon deux classes :
- courriel pertinent ;
- pourriel intempestif.

Supposons que le classificateur soit testé avec un jeu de 200 mails, dont 100 sont des courriels pertinents et les 100 autres relèvent de pourriels.

Pour cela, on veut savoir :

- combien de courriels seront faussement estimés comme des pourriels (fausses alarmes) par le classificateur ;
- combien de pourriels ne seront pas estimés comme tels (non détections) et classifiés à tort comme courriels par le classificateur.

A l'aide de données étiquetées par un expert humain et du programme kNN, on dresse le tableau suivant :

		Classe estimée (programme classificateur kNN)	
		Courriel	**Pourriel**
Classe réelle (vérificateur humain)	**Courriel (100)**	**95** Vrais positifs	5 Faux négatifs
	Pourriel (100)	3 Faux positifs	**97** Vrais négatifs

Matrice de confusion

La matrice de confusion ci-dessus se lit alors comme suit :

- horizontalement, sur les 100 courriels initiaux (ie : 95+5), 95 ont été estimés par le classificateur comme tels et 5 ont été estimés comme pourriels (ie : 5 **faux-négatifs**) ;
- horizontalement, sur les 100 pourriels initiaux (ie : 3+97), 3 ont été estimés par le classificateur comme courriels (ie : 3 **faux-positifs**) et 97 ont été estimés comme pourriels ;
- verticalement, sur les 98 mails (ie : 95+3) estimés par le classificateur comme courriels, 3 sont en fait des pourriels ;
- verticalement, sur les 102 mails (ie : 5+97) estimés par le classificateur comme pourriels, 5 sont en fait des courriels ;
- diagonalement (du haut gauche, au bas droit), sur les 200 courriels initiaux, 192 (95 + 97) ont été estimés correctement par le classificateur.

Cette notion s'étend à un nombre quelconque de classes.
On peut normaliser cette matrice pour en simplifier la lecture : dans ce cas, un classificateur sera d'autant meilleur que sa matrice de confusion s'approchera d'une matrice diagonale.

Le script précédent était destiné à illustrer l'algorithme des k plus proches voisins : seul un point test était

envisagé.
On cherche ici à évaluer les performances de cet algorithme à travers un jeu de données étiquetées par un « expert ». Les performances de l'algorithme sont évaluées grâce à la matrice de confusion, créée en comparant prédictions et données étiquetées.

A partir d'un ensemble de données étiquetées par un expert, on souhaite créer une partition constituée :

- d'un ensemble d'apprentissage A ;
- d'un ensemble test.

Objectif ✔️ : évaluer les performances de l'algorithme kNN en comparant l'étiquetage fourni par la fonction kNN pour les données de l'ensemble test à l'étiquetage expert.

Etape 1 - Partition des données expert en données d'apprentissage et données test

Expérimentation

🛠️ 👁️ La fonction partitionAT(pcA, dataE, targetE) (disponible dans l'une des bibliothèques) construit l'ensemble d'apprentissage A en prélevant un pourcentage, noté pcA ($0 \leq pcA < 1$), d'objets dans les données expert ; l'ensemble test T est constitué des objets restants.

Comparaison des prévisions pour un point prélevé au hasard dans l'ensemble test : exécuter la cellule plusieurs fois afin de voir apparaître une différence entre classe estimée et classe expert.

[ ]:
```
# Partition des données étiquetées : ensemble d'apprentissage et ensemble test
A, T = partitionAT(20/100, data, target)
dataA, targetA = A   # Données et étiquettes expert ensemble d'apprentissage
dataT, targetT = T   # Données et étiquettes expert ensemble test

# Tirage au hasard d'un élément de l'ensemble de test
idx = randint(0, len(dataT)-1)  # index aléatoire
o = dataT[idx]                  # Objet test
e_Exp = targetT[idx]            # Etiquette Expert

# Estimation de l'étiquette à l'aide de l'algorithme kNN
e_kNN = kNN(o, dataA, targetA, 5, d_Euclide)

# Comparaison étiquette estimée/étiquette expert
print(f"Etiquette estimée : {e_kNN} \nEtiquette expert  : {e_Exp}")
```

Etape 2 - Classification des données test

💡 Observer le code de la fonction kNNclassification(A, T, k, fct_distance) qui permet de déterminer l'étiquette estimée pour toutes les données de l'ensemble test T.

💻 Exécuter le code.

[ ]:
```
def kNNclassification(A, T, k, fct_distance):
    """
    Paramètres :
        A : de la forme (dataA, targetA) ensemble d'apprentissage
        T : de la forme (dataT, targetT) ensemble de test
        k : nombre de plus proches voisins à considérer
        fct_distance : fonction utilisée pour calculer la distance
    Renvoie :
        liste des classes estimées pour les données de T par l'algorithme kNN
    """
    dataA, targetA = A  # Données et étiquettes expert ensemble d'apprentissage
    dataT, targetT = T  # Données et étiquettes expert ensemble test
    return [kNN(o, dataA, targetA, k, fct_distance) for o in dataT]
```

[ ]:
```
# Etiquettes de l'ensemble test T déterminées par l'algorithme kNN
e_kNN = kNNclassification(A, T, 3, d_Euclide)

# Comparaison étiquettes estimées/étiquettes expert
print(f"Etiquettes estimées :\n{e_kNN} \nEtiquettes expert : \n{targetT}")
```

La matrice de confusion permet la comparaison des étiquetages de façon plus parlante.

Etape 3 - Matrice de confusion

✏️ *list.index(valeur)*

L.index(x) Renvoie l'index de la valeur x dans la liste L si x est dans L (erreur sinon).

[ ]:
```
# Exemples
L = [8, -3, 2]
print(L.index(-3))
print(L.index(0))
```

💡 Principe de construction de la matrice de confusion.

On dispose pour construire cette matrice de confusion mat_conf de deux listes d'étiquettes e_Exp et e_kNN et d'une liste classes représentant les classes.

Par exemple :
e_Exp = ['b', 'r', 'y', 'r', ...], e_kNN = ['b', 'y', 'y', 'r', ...] et classes = ['r', 'b', 'y'].

Pour remplir la matrice, il faut parcourir la liste des étiquettes estimées (index k dans le code ci-dessous).
Dans cet exemple, au rang $k = 1$, on a e_exp[k=1]='r' et e_exp[k=1]='y' : comment doit-on tenir compte de ces deux informations dans la matrice ?
- l'étiquette experte donne la ligne i de la matrice, c'est le rang (l'index) de la valeur 'r' dans la liste des classes (donc 0 ici) ;
- l'étiquette estimée donne la colonne j de la matrice, c'est le rang (l'index) de la valeur 'y' dans la liste des classes (donc 2 ici).
Il suffit donc d'ajouter la valeur 1 au coefficient mat_conf[i=0][j=2] pour ce couple d'étiquettes expert/estimée.

✏️ Commenter le code ci-dessous.

[ ]:
```
def confusion(e_Exp, e_kNN, classes):
    """
    Paramètres :
        e_Exp :   étiquettes expert
        e_kNN :   étiquettes estimées par l'algorithme kNN
        classes : classes du jeu de données (apprentissage et test)
    Renvoie :
        mat_conf : matrice de confusion
    """
    #
    n = len(classes)
    #
    mat_conf = [[0 for i in range(n)] for j in range(n)]
    #
    for k in range(len(e_kNN)):      # k =
        i = classes.index(e_Exp[k]) # i =
        j = classes.index(e_kNN[k]) # j =
        mat_conf[i][j] += 1         #
    # Affichage
    for l in range(n):
        print(mat_conf[l])
    return mat_conf
```

[ ]:
```
# Matrice de confusion
mc = confusion(targetT, e_kNN, classes)
```

4 Expérimentation

4.1 Fonction pour la représentation graphique

La fonction nuageAT(A, T, e_kNN, infos='') permet de représenter les données étiquetées :
- petits cercles de couleur = données test étiquetées par l'expert ;
- carrés foncés = données d'apprentissage étiquetées par l'expert ;
- carrés clairs = prévisions de l'algorithme kNN.

En cliquant sur le bouton SVG sous le graphe, il est possible de télécharger et de visualiser (Firefox ou Chrome) le graphe et de l'agrandir afin d'observer étiquettes expert et étiquettes prévues par l'algorithme.

4.2 💡 Programme complet

🛠️ 👁️ Modifier la valeur de la « graine » du générateur pseudo-aléatoire seed à plusieurs reprises et visualiser les résultats.

[ ]:
```
#________Programme complet________

# Initialisation générateur nombres pseudo-aléatoires = > répétabilité
seed(1) # Modifier la valeur entre parenthèses pour modifier le tirage

# Ensemble d'apprentissage
n = 500                        # Nombre d'objets (vecteurs 2D représentés par des points)
classes = ['r', 'b', 'y']      # Classe = couleur des points

# Création du jeu de données : objets et étiquettes associées
data, target = tirage_gauss(n, classes, mu=200)

# Partition des données étiquetées : ensemble d'apprentissage et ensemble test
pcA = 0.2 # Pourcentage de points dans l'ensemble d'apprentissage
A, T = partitionAT(pcA, data, target)
dataA, targetA = A
dataT, targetT = T

# Etiquettes de l'ensemble test T déterminées par l'algorithme kNN
k = 3
e_kNN = kNNclassification(A, T, k, d_Euclide)

# Matrice de confusion
print("Matrice de confusion :")
mc = confusion(targetT, e_kNN, classes)

# Graphe
txt = f"k = {k} plus proches voisins avec {n} objets"
txt += f"\n Test : {n-int(pcA*n)} objets - Apprentissage : {int(pcA*n)} objets"
nuageAT(A, T, e_kNN, infos=txt)
```

5 Bibliothèque scikit learn

👁 *scikit learn*

L'algorithme des k plus proches voisins est implémenté via un classificateur :
sklearn.neighbors.KNeighborsClassifier
Objet classificateur :
clf = sklearn.neighbors.KNeighborsClassifier(n_neighbors=k)

Définition des données d'apprentissage (objets et étiquettes) :
clf.fit(data, target)

Etiquettes calculées avec des données test ou nouvelles données :
clf.predict(data)

Probabilités d'appartenance à une classe :
clf.predict_proba(data)

Frontières entre domaines de prédominance des différentes classes :
sklearn.inspection.DecisionBoundaryDisplay.from_estimator(clf, data,...)

6 Bibliothèque scikit learn - Application à l'OCR

Cet exemple est tiré de la bibliothèque scikit learn (informations détaillées : Recognizing hand-written digits).

Les données (images 8x8 pixels noir et blanc), leurs étiquettes (chiffres associés) ainsi que d'autres renseignements sont stockés dans un dataset.
Le chargement de ces données s'effectue grâce à la commande datasets.load_digits() (syntaxe détaillée : sklearn.datasets.load_digits).

OCR - Etape 1- Chargement des bibliothèques

[ ]:
```
# Bibliothèques
from sklearn import datasets      # Exemples de données (datasets) disponibles
from sklearn.neighbors import KNeighborsClassifier   # Algorithme kNN
```

OCR - Etape 2 - Chargement du dataset

Dataset : tableaux data (images) et target (étiquettes) constituant l'ensemble d'apprentissage A.

[ ]:
```
# Chargement du dataset pour l'OCR (objets images et étiquettes associées)
digits = datasets.load_digits()

# data = tableau des images, target = tableau des étiquettes
data, target = digits.data, digits.target
```

💡 👁 **Visualisation des données**

[ ]:
```
# Structure du tableau contenant les images
NbreImages, dimImage = data.shape
print(f"Nombre d'images dans data : {NbreImages} \nTaille de l'objet (vecteur) image :␣
 ↪{dimImage} \n")

# "Vecteur" correspondant à la première image du tableau data
vect0 = data[0]
print(f"Vecteur image à {dimImage} coordonnées : \n{vect0}\n")

# Reconstitution de la "matrice" image 8x8
image0 = vect0.reshape(8,8)
print(f"Matrice image 8x8 pour la 1ère image : \n{image0}\n")
```

```
# Affichage de l'étiquette de l'image
print(f"Etiquette de la 1ère image : {target[10]}\n")

# Classes
print(f"Classes : {digits.target_names}")

# Affichage de l'image
plt.figure()
plt.set_cmap('gray_r')
plt.imshow(data[0].reshape(8,8))
plt.show()
```

OCR - Etape 3 - Utilisation de kNNclassification (paragraphe I)

Récupération des classes : digits.target_names (cf. ci-dessus)

[ ]:
```
classes = digits.target_names
classes
```

Partition du jeu de données étiquetées en un ensemble d'apprentissage et un ensemble test grâce à partitionAT :

💡 Les conversions array → list ci-dessous sont uniquement destinées à assurer la compatibilité avec les fonctions programmées dans les paragraphes précédents.

[ ]:
```
pcA = 0.2 # Pourcentage de points dans l'ensemble d'apprentissage
classes = list(classes)
A, T = partitionAT(pcA, list(data), list(target))
dataA, targetA = A
dataT, targetT = T
print("1ère image de l'ensemble d'apprentisage :\n", A[0][0])
```

Exécution de kNNclassification
⚠️ ATTENTION temps d'exécution un peu long ...

[ ]:
```
# Etiquettes de l'ensemble test T déterminées par l'algorithme kNN
k = 3
e_kNN = kNNclassification(A, T, k, d_Euclide)
print("Etiquettes calculées (10 premières) :\n", e_kNN[:10])
```

Matrice de confusion : confusion (cf. ci-dessus)

[ ]:
```
# Matrice de confusion
mc = confusion(targetT, e_kNN, classes)
```

OCR - Etape 4 - Utilisation de la fonction KNeighborsClassifier de sklearn

[ ]:
```
k=3

# Initialisation d'un 'classificateur'
knn = KNeighborsClassifier(n_neighbors=k).fit(dataA, targetA)

# Récupération des étiquettes calculées
e_sklearn = knn.predict(dataT)

# Matrice de confusion (fonction programmée ci-dessus)
mc = confusion(targetT, e_sklearn, classes)

print("\nEtiquettes calculées (10 premières) :\n", e_sklearn[:10])
```

👁️Comparer les matrices de confusion obtenues ci-dessus par la fonction programmée et celle de sklearn.

7 Bibliothèque scikit learn - Domaines de prédominance

Exemple tiré de : Iris Dataset

Le jeu de données comprend 50 échantillons de chacune des trois espèces d'iris (Iris setosa, Iris virginica et Iris versicolor).
Quatre caractéristiques ont été mesurées à partir de chaque échantillon : la longueur et la largeur des sépales et des pétales, en centimètres (Iris de Fisher (Wikipédia)).

Les données sont regroupées dans un tableau 150 x 4 :
- les lignes correspondent aux échantillons ;
- les colonnes correspondent aux caractéristiques (Sepal Length, Sepal Width, Petal Length and Petal Width).

Iris - Chargement des bibliothèques et caractéristiques du dataset

[ ]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets
from sklearn.inspection import DecisionBoundaryDisplay
```

[ ]:
```
# Chargement des données
iris = datasets.load_iris()

# Données relatives aux iris : 4 caractéristiques pour 3 variétés
#print(iris.DESCR, '\n')

# 4 caractéristiques des iris
print('4 caractéristiques des iris : \n', iris.feature_names, '\n')

# Noms des 3 variétés d'iris
print("Noms des 3 variatés d'iris : \n", iris.target_names, '\n')

# Données d'apprentissage = training data (150 données)
X = iris.data[:, :2]    # 2 caractéristiques retenues sur les 4 disponibles
print('Données = 2 caractéristiques (5 1ères lignes) : \n', iris.data[:5], '\n')

# Etiquetage expert (3 classes d'iris = 3x50 étiquettes)
y = iris.target
print('Etiquetage expert (3 classes) : \n', iris.target, '\n')
```

Iris - Probabilités d'appartenance à une classe

[ ]:
```
# Nombre de plus proches voisins pris en compte
k = 15

# Initialisation d'un 'classificateur' nommé clf
clf = neighbors.KNeighborsClassifier(n_neighbors=k)
clf.fit(X, y)

# Prédiction de la classe associée à des données non étiquetées
point1 = [4.5, 3.]
point2 = [8.0, 2.0]
points = [point1, point2]
classes = clf.predict(points)
print('Classes : ', classes, '\n')

# Noms des espèces associés à la classe
noms = iris.target_names[classes]
print('Variétés : ', noms, '\n')
```

```
# Estimation de la probabilité liée à la détermination de la classe
# Renvoie un array avec la probabilité d'appartenance à chaque classe
probas = clf.predict_proba(points)
print('Probabilités : \n', probas, '\n\n')

# Affichage plus explicite
for pt, cl, nom, pr in zip(points, classes, noms, probas):
    print(f'Point {pt} : variété {nom} (classe {cl}) \n avec les probabilités :\
     \n{pr[0]*100} % pour Setosa,\
     \n{pr[1]*100} % pour Versicolor,\
     \n{pr[2]*100} % pour Virginica \n')
```

Iris - Données concernant les sépales

[ ]:
```
# Nuage des points d'apprentissage avec couleur associée à la classe
plt.figure()
dic_coul = {0: 'darkorange', 1: 'c', 2:'darkblue'}
variete = {0: 'Setosa', 1: 'Versicolor', 2:'Virginica'}
coul = [dic_coul[c] for c in y]
for i in dic_coul:
    plt.scatter(
        x=X[i*50:(i+1)*50, 0],
        y=X[i*50:(i+1)*50, 1],
        c=coul[i*50:(i+1)*50],
        alpha=1.0,
        edgecolor="black",
        label=variete[i],
    )
plt.xlabel("Longueur des sépales (cm)")
plt.ylabel("Largeur des sépales (cm)")
plt.legend()
plt.title(f"3 classes d'iris - k = {k} - Données sépales")
plt.show()
```

Iris - Frontières entre domaines de prédominance

[ ]:
```
# Nombre de plus proches voisins pris en compte
n_neighbors = 15

# Données
iris = datasets.load_iris()

# Données d'apprentissage (2 caractéristiques des sépales)
X = iris.data[:, :2]

# Etiquetage expert (3 classes d'iris)
y = iris.target

# 3 couleurs pour les domaines des 3 classes d'iris
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])

# Utilisation de 2 critères différents de pondération des données :
# -> 'uniform' = tous les points du voisinage ont le même poids
# -> 'distance' = poids proportionnel à l'inverse de la distance (poids d'autant plus grand que␣
 ↪les points sont proches)
weights = "uniform"

# Classificateur
clf = neighbors.KNeighborsClassifier(n_neighbors,  weights=weights)
clf.fit(X, y)
```

```
# Création d'un graphique
plt.figure()

# Choix auto des points tests et détermination de la classe
DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    cmap=cmap_light,
    response_method="predict",
    plot_method="pcolormesh",
    xlabel=iris.feature_names[0],
    ylabel=iris.feature_names[1],
    shading="auto",
)
# Nuage de points (données sépales)
dic_coul = {0: 'darkorange', 1: 'c', 2:'darkblue'}
variete = {0: 'Setosa', 1: 'Versicolor', 2:'Virginica'}
coul = [dic_coul[c] for c in y]
for i in dic_coul:
    plt.scatter(
        x=X[i*50:(i+1)*50, 0],
        y=X[i*50:(i+1)*50, 1],
        c=coul[i*50:(i+1)*50],
        alpha=1.0,
        edgecolor="black",
        label=variete[i],
    )
plt.xlabel("Longueur des sépales (cm)")
plt.ylabel("Largeur des sépales (cm)")
plt.legend()
plt.title(f"3 classes d'iris - k = {n_neighbors} - Frontières entre classes")
plt.show()
```