```
import random
import numpy as np

#Changer la direction du fichier
f = open(r'C:\Users\Elie\Downloads\data.csv')
f2 = open(r'C:\Users\Elie\Downloads\preTest.csv')
ftest = open(r'C:\Users\Elie\Downloads\finalTest.csv')
#Nombres de voisins désirés
k = 3
#Pourcentage de réussite de l'algorythme
reussite = 88
#Puissance distance
p = 2

#Sortie brute des données
data1, data2, datatest = [], [], []

#Lecture du fichier et des données
for line in f:
    ligne, ajout = line.split(','), []
    if(len(ligne)>2):
        for i in range(len(ligne)-1):
            ajout.append(float(ligne[i]))
        ajout.append(ligne[len(ligne)-1].strip('\n'))
        data1.append(ajout)
f.close()

for line in f2:
    ligne, ajout = line.split(','), []
    if(len(ligne)>2):
        for i in range(len(ligne)-1):
            ajout.append(float(ligne[i]))
        ajout.append(ligne[len(ligne)-1].strip('\n'))
        data2.append(ajout)
f2.close()

for line in ftest:
    ligne, ajout = line.split(','), []
    if(len(ligne)>2):
        for i in range(len(ligne)-1):
            ajout.append(float(ligne[i]))
        ajout.append(ligne[len(ligne)-1].strip('\n'))
        datatest.append(ajout)
ftest.close()

def get_type(elt):
    return elt[6]

data = data1 + data2
data.sort(key=get_type)
data2.sort(key=get_type)

"""
Foncion qui prend une liste en argument et retourne des listes ne
comportant qu'un seul type
Les éléments de la liste d'entrée doivent être triés de manière à se
suivre selon leurs types
"""
def Split(liste):
    retour, temp = [], []
```

```
    for i in liste:
        if(len(temp)==0):
            temp.append(i)
        elif(temp[-1][6]==i[6]):
            temp.append(i)
        else:
            retour.append(temp)
            temp = []
            temp.append(i)
    retour.append(temp)
    return retour

"""
Fonction qui permet de séparer une liste d'un type en deux nouvelles
listes: une pour l'apprentissage et une pour la validation
"""
def Repartition(liste):
    copie, compteur, apprentissage, confirmation = liste.copy(), 1, [], []
    while(len(copie)!=0): #Tant que des fleurs ne sont pas assigner
        #On ajoute une fleur à l'apprentissage
        if(compteur%2==0 and len(copie)!=0):
            i = random.randint(0,len(copie)-1)
            apprentissage.append(copie[i])
            copie.remove(copie[i])
            compteur = compteur + 1
        #On ajoute ensuite une fleur à la confirmation
        if(compteur%2!=0 and len(copie)!=0):
            i = random.randint(0,len(copie)-1)
            confirmation.append(copie[i])
            copie.remove(copie[i])
            compteur = compteur + 1
    return [apprentissage, confirmation]

"""
Fonction qui permet à partir d'une liste rangées selon les type de créer
deux listes égales (si possible) avec une répartition égales (si possible)
une pour l'apprentissage et une pour la validation comportant tout les
types
"""
def ListeFinale(liste):
    apprent, conf = [], []
    for i in range(len(liste)): #Pour chaque type de fleur
        #On répartit les fleurs de ce type en deux listes
        split = Repartition(liste[i])
        apprent, conf = apprent + split[0], conf + split[1]
    return apprent, conf

"""
Fonction qui permet de trier les données selon leur distance croissante
"""
def Ordination(liste): # Ordination des données en fonction des distances
    triee = sorted(liste)
    return [i[1] for i in triee]

"""
Fonction permettant de calculer la distance euclidienne
"""
def Dist(a,b): # calcul de la distance euclidienne
    return ((abs(a-b))**p)**(1/p)

"""
```

```
Fonction permettant de calculer la distance de l'élément de test avec
chaque élément de la base de donnée apprise
"""
def Distance(elt):
    dist = []
    for i in base: #Changer les arguments de calcul des distances
        d = .0
        for j in range(len(i)-1):
            d = d + Dist(float(i[j]),float(elt[j]))
        d = d / 6 #On fait la moyenne des distances sur chacun des critères
        dist.append([d, i])
    return dist

"""
Fonction qui permet de relever la fréquence d'apparition de chaque
type dans une liste classée de manière décroissante
"""
def Frequence(liste):
    frequence, typefreq = [], []
    for i in range (len(liste)):
        #On vérifie si on a déja rencontré ce type
        if(liste[i][6] in typefreq):
            index = typefreq.index(liste[i][6])
            frequence[index] = frequence[index] + 1
        #On vérifie si c'est la première fois que l'on rencontre ce type
        if(liste[i][6] not in typefreq):
            frequence.append(1)
            typefreq.append(liste[i][6])
    retour = []
    for i in range(len(frequence)):
        retour.append([frequence[i],typefreq[i]])
    #Tri en ordre de fréquence décroissante
    retourtri = sorted(retour, reverse = True)
    return retourtri[0]

"""
Fonction qui permet de créer la base de données apprises qui vont servir
à la reconnaissance des tests
"""
def Apprentissage(listeapprentissage):
    global base
    base = listeapprentissage

"""
Fonction qui permet à partir d'une liste, dont ont connait le type de
chaque élément, de retourner le pourcentage de bonne déduction de
l'algorythme
"""
def Confirmation(liste):
    global confusion
    confusion = np.zeros((5,5))
    #Pour chaque élément i de la liste de confirmation
    for i in liste:
        sol = KNN(i)
        MatriceConfusion(sol, i[6])
    #On retourne le pourcentage de bonne déduction
    pourcentage = 0
    for i in range(5):
        pourcentage = pourcentage + confusion[i,i]
    return (pourcentage * 100)/len(liste)
```

```
def Ligne(classe):
    switcher = {
        'classA': 0,
        'classB': 1,
        'classC': 2,
        'classD': 3,
        'classE': 4,
        }
    return switcher.get(classe)

def Colonne(classe):
    switcher = {
        'classA': 0,
        'classB': 1,
        'classC': 2,
        'classD': 3,
        'classE': 4,
        }
    return switcher.get(classe)

def MatriceConfusion(sol, elt):
    i,j = Ligne(elt), Colonne(sol)
    confusion[i,j] = confusion[i,j] + 1

"""
Fonction qui permet de réaliser l'apprentissage et la confirmation
de l'algorythme avec le taux d'erreur demandé
"""
def Initialisation(liste):
    split = Split(liste)
    datalearning,datavalidation = ListeFinale(split)
    Apprentissage(datalearning)
    pourcentagereussite = Confirmation(datavalidation)
    while(pourcentagereussite<reussite):
        split = Split(liste)
        datalearning, datavalidation = ListeFinale(split)
        Apprentissage(datalearning)
        pourcentagereussite = Confirmation(datavalidation)
    return pourcentagereussite

"""
Fonction qui permet de faire la connection entre toute les autres
fonctions
"""
def KNN(elt): # Programme principal de la méthode KNN
    ListeDistance = Distance(elt)
    ordonnee, top  = Ordination(ListeDistance), []
    for i in range(k):
        top.append(ordonnee[i])
    return Frequence(top)[1]

if __name__ == '__main__' :
    Initialisation(data)
    r = open(r'C:\Users\Elie\Downloads\Obadia_Samples.txt','w')
    for i in datatest:
        r.write(KNN(i)+('\n'))
    r.close()
```