RAPPORT IA – CLASSIFICATION CHALLENGE

LE CODE

CALCULER LA DISTANCE EUCLIDIENNE

La première étape dans le programme est de calculer la distance euclidienne entre deux points (donc entre chaque coordonnée entre chaque ligne du dataset). Plus la distance est faible, plus les points sont similaires et sont donc probablement de même label.

On peut noter qu'on aurait pu utiliser la distance Manhattan mais celle-ci prend plus de temps d'exécution après l'avoir testé : 6m48 contre 5m41 pour les prédictions de FinalTest.

```
def distance_euclidienne(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

def distance_manhattan(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance+=abs(row1[i]-row2[i])
    return distance
```

RECUPERER LES VOISINS LES PLUS PROCHES

Les voisins d'un point sont ses k plus proches points selon la distance euclidienne définie dans la section précédente. On calcule la distance euclidienne entre le point à tester (test_row) et tous les points de train (la dataframe d'apprentissage).

Une fois toutes les distances calculées et récupérées dans une liste « distances », on les range par ordre croissant pour récupérer les k premiers voisins.

```
72      def get_voisins(train, test_row, k):
73          distances = []
74          for train_row in train.iloc: #On parcourt les points de notre dataframe train
75              distance = distance_euclidienne(test_row, train_row) #On calcule la distance
76              distances.append((train_row, distance))
77          distances.sort(key=lambda t: t[1])  #On range les distances par ordre croissant
78          voisins = []
79          for i in range(0,k):
80              voisins.append(distances[i][0])  #On prend les k premiers voisins (point) que
81          return voisins
82
```

ANALYSE DES VOISINS

Méthode qui crée une prédiction du label d'un point à partir du label de ses voisins. La prédiction sera le label le plus représenté parmi les k plus proches voisins du point.

On s'aide donc du dataset d'apprentissage train car on connait les labels.

```
classes = ['classA','classB','classC','classD','classE']

def analyse_voisins(train,test_row,k):
    occ=[] #occ = occurence pour chaque classe
    voisins = get_voisins(train,test_row,k) #Récup la liste des k voisins de test_row
    list_label = [a[6] for a in voisins] #Recup la liste des labels des voisinns
    for classe in classes:
        occ.append(list_label.count(classe))  #Remplit la liste des occurences => le nombre de fois
    prediction = classes[occ.index(max(occ))] #récupère la classe la mieux représentée parmis la li
    return prediction
```

ALGORITHME KNN

Méthode qui va créer la liste des prédictions pour la dataframe test dont on ne connait pas les labels à l'aide du dataset d'apprentissage. C'est en soit la méthode analyse_voisins appliquée pour chaque point de test.

```
def Knn(train, test, k):
    predictions = list()
    for row in test.iloc:
        prediction = analyse_voisins(train, row, k)
        predictions.append(prediction)
    return(predictions)
```

PRECISION DES PREDICTIONS POUR UN K DONNE

Méthode qui récupère la précision des prédictions d'un data set d'apprentissage. C'est un counter qui prend +1 pour chaque similarité entre la prédiction et le label du point de data.

```
def get_precision(data,predictions):
    a=0
    count=0
    for prediction in predictions: #parcourt les prédictions
        if prediction == data.iloc[a][6]: #Compare la prédiction à la classe actuelle du point
            count+=1
        a+=1
    return round(count / float(len(data))*100,2) #arrondit à 2 chiffres après la virgule
```

EVALUATION DES K

Méthode qui évalue chaque la précision des prédictions pour chaque k dans un intervalle que l'on choisit.

Pour cela on utilise la méthode Knn et la méthode get_precision pour chaque k de l'intervalle. Cette méthode nous permet de déterminer quel k choisir pour le test final.

```
k_range = range(1,101)

def evaluation_des_k(train,test,k_range):
    for k in k_range:
        predictions = Knn(train,test,k)
        a = "précision pour k = {} : {}%".format(k,get_precision(test,predictions))
        print(a)
```

SEPARATION DE DATAFRAME EN 2 SOUS PARTIES

Sépare la dataframe data en deux sous data frames TRAIN et TEST en choisissant le pourcentage de données dans TRAIN par rapport à DATA Dans ce code, j'ai choisi n = 0.8 tout le temps mais j'ai parfois testé n pour 0.9 ou 0.5 pour diminuer le temps d'exécutions du code.

```
def split(data,n):
    taille_pli = int(len(data)*n)
    df1 = data.iloc[:taille_pli,:]
    df2 = data.iloc[taille_pli:,:]
    return df1,df2
```

RECUPERATION DES DATAFRAMES

J'ai utilisé la librairie pandas pour récupérer les dataframes de la façon suivante :

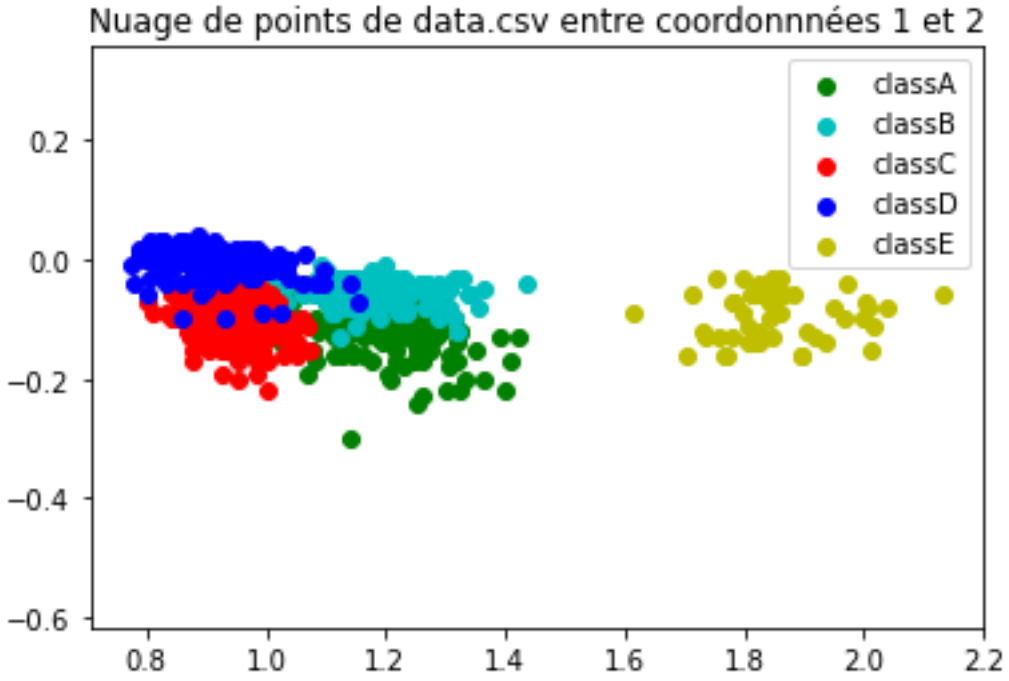
```
data_learn = pd.read_csv('data.csv',header = None)
data_test = pd.read_csv('pretest.csv',header = None)
#data_learn2 = data_learn.append(data_test,sort=False) # data.csv + pretest.csv
data_testFinal = pd.read_csv('finalTest.csv',header = None)
```

CHOIX DU K

Avant tout, j'ai commencé par tracer des plots des points par couple de coordonnées pour me donner une idée de la dispersion des points par classe. Pour cela j'ai effectué le code suivant pour data.csv :

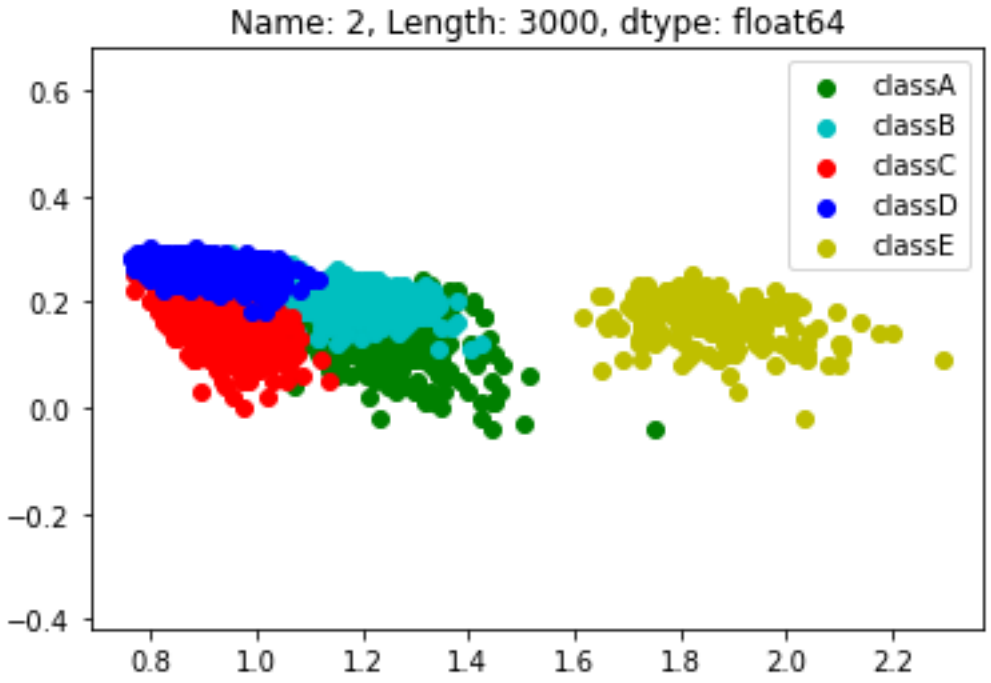
```
def plot(filename):
    for i in range(0,6):
        for j in range(0,6):
            if i != j:
                x = filename.loc[:,i]
                y = filename.loc[:,j]
                lab = filename.loc[:,6]
                plt.axis('equal')
                plt.scatter(x[lab == 'classA'], y[lab == 'classA'], color='g', label='classA')
                plt.scatter(x[lab == 'classB'], y[lab == 'classB'], color='c', label='classB')
                plt.scatter(x[lab == 'classC'], y[lab == 'classC'], color='r', label='classC')
                plt.scatter(x[lab == 'classD'], y[lab == 'classD'], color='b', label='classD')
                plt.scatter(x[lab == 'classE'], y[lab == 'classE'], color='y', label='classE')
                plt.title("Nuage de points de data.csv entre coordonnnées {} et {}".format(x,y))
                plt.legend()
                plt.show()
```

Un graphe qui m'a bien fait comprendre le problème est celui-là par exemple :

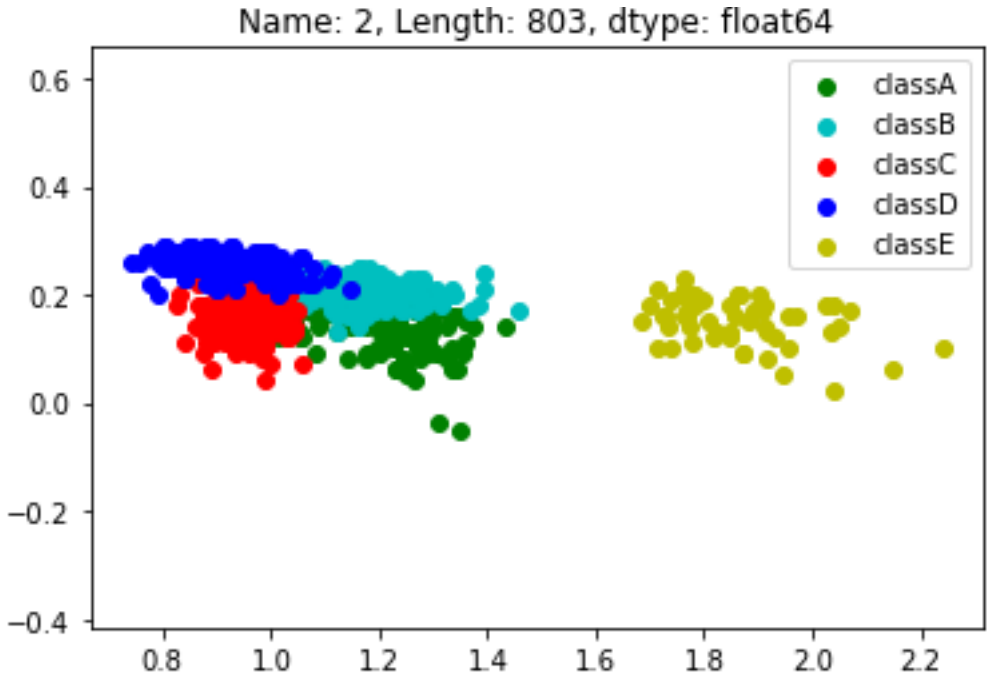


On voit bien qu'il va y avoir des conflits entre les classes A, B, C et D mais pour la classe E, on devrait avoir des résultats cohérents peu importe le K que l'on prend.

Même chose pour le prétest, on trouve des graphs comme :



Même chose avec les finalTest après avoir estimé les classes :



Cependant, pour les predictions obtenues avec la dataframe finalTest, on voit bien que pretest.csv se rapproche plus de finalTest.csv que data.csv.

Data.csv n'est en fait qu'une translation de 0,5 de FinalTest.csv sur l'axe des ordonnés sur le graphe.

Cela va influencer mon choix de k en privilégiant pretest.csv à data.csv.

Pour choisir mon k, j'ai d'abord entrainé mon algorithme avec le dataset data.csv où je prenais 80% des données en entrainement et les 20 % restant en test à l'aide de la méthode **split(data, n).**

A l'aide de la méthode **evaluation_des _k(train, test, k_range)** , j'ai testé la précision de chaque k dans une range que j'ai déterminé de 1 à 25 car au-dessus, je tombais en sur-apprentissage.

Voici ce que ça m'a donné :

```
On va donc le tester pour les autres datas pour vérifier sa qualité.

Avec data.csv que l'on split en deux parties (80% train, 20% test):
train,test = split(data_learn,0.8)
evaluation_des_k(train,test,k_range)

    précision pour k = 1 : 91.93%
    précision pour k = 2 : 87.58%
    précision pour k = 3 : 90.68%
    précision pour k = 4 : 91.93%
    précision pour k = 5 : 92.55%
    précision pour k = 6 : 93.79%
    précision pour k = 7 : 93.17%
    précision pour k = 8 : 93.17%
    précision pour k = 9 : 93.17%
    précision pour k = 10 : 91.93%
    précision pour k = 11 : 91.93%
    précision pour k = 12 : 91.3%
    précision pour k = 13 : 90.06%
    précision pour k = 14 : 91.93%
    précision pour k = 15 : 90.06%
    précision pour k = 16 : 90.68%
    précision pour k = 17 : 88.82%
    précision pour k = 18 : 89.44%
    précision pour k = 19 : 89.44%
    précision pour k = 20 : 90.06%
    précision pour k = 21 : 89.44%
    précision pour k = 22 : 90.06%
    précision pour k = 23 : 87.58%
    précision pour k = 24 : 87.58%
    précision pour k = 25 : 86.96%
temps d'exécution: 6:37

On remarque bien que le meilleur k ici est 6 -> 93,79 %
```

Le meilleur k est 6 avec une précision de 93,79 %. Donc j'ai commencé par vouloir utiliser k = 6 pour le finalTest. Cependant, comme indiqué avant, data ne représente pas bien finalTest, je décide de mélanger les deux pour voir :

On fait la même chose avec train = data.csv et test = pretest.csv pour comparer les valeurs :

```
evaluation_des_k(data_learn,data_test,k_range)


Résultats pour TRAIN = data.csv et TEST = preset.csv :
    précision pour k = 1 : 68.24%
    précision pour k = 2 : 61.02%
    précision pour k = 3 : 68.99%
    précision pour k = 4 : 68.49%
    précision pour k = 5 : 69.24%
    précision pour k = 6 : 69.49%
    précision pour k = 7 : 70.49%
    précision pour k = 8 : 70.61%
    précision pour k = 9 : 71.98%
    précision pour k = 10 : 72.1%
    précision pour k = 11 : 72.85%
    précision pour k = 12 : 72.85%
    précision pour k = 13 : 73.85%
    précision pour k = 14 : 72.35%
    précision pour k = 15 : 73.6%
    précision pour k = 16 : 72.48%
    précision pour k = 17 : 73.6%
    précision pour k = 18 : 72.98%
    précision pour k = 19 : 73.1%
    précision pour k = 20 : 72.85%
    précision pour k = 21 : 73.1%
    précision pour k = 22 : 72.6%
    précision pour k = 23 : 73.47%
    précision pour k = 24 : 73.35%
    précision pour k = 25 : 73.6%
temps d'éxécution : 47:12

On remarque bien que le meilleur k ici est 13 -> 73,85 %
```

On trouve k = 13 avec 73,85%.

Après quelques recherches sur internet sur la meilleure façon de choisir son k, j'ai lu que k est souvent impair, non multiple de n=nombre de tuples donc = 802 et parfois un bon k est (racine de n) /2 ce qui vaut 13,5.

13 paraissait donc un bon choix mais comme preTest.csv est plus proche de finalTest.csv que data.csv, j'ai décidé de refaire une dernière fois avec train = 80% de pretest.csv et test = 20% de pretest.csv.

Voici les résultats :

```
Avec pretest.csv que l'on split en deux parties (80% train, 20% test):
train,test = split(data_test,0.8)
evaluation_des_k(train,test,k_range)

    précision pour k = 1 : 83.85%
    précision pour k = 2 : 81.99%
    précision pour k = 3 : 90.68%
    précision pour k = 4 : 90.06%
    précision pour k = 5 : 90.68%
    précision pour k = 6 : 91.3%
    précision pour k = 7 : 89.44%
    précision pour k = 8 : 92.55%
    précision pour k = 9 : 91.3%
    précision pour k = 10 : 91.3%
    précision pour k = 11 : 88.82%
    précision pour k = 12 : 87.58%
    précision pour k = 13 : 87.58%
    précision pour k = 14 : 88.2%
    précision pour k = 15 : 87.58%
    précision pour k = 16 : 86.96%
    précision pour k = 17 : 85.71%
    précision pour k = 18 : 86.96%
    précision pour k = 19 : 85.09%
    précision pour k = 20 : 85.09%
    précision pour k = 21 : 84.47%
    précision pour k = 22 : 83.85%
    précision pour k = 23 : 83.85%
    précision pour k = 24 : 84.47%
    précision pour k = 25 : 83.85%
temps d'exécution: 6:41

On remarque bien que le meilleur k ici est 8 -> 92,55 %


""""
```

Mon k sera finalement 8 car il a une précision de 92,55 %. Il a également une bonne précision pour les autres (70,61% et 93,17%).

Avec notre k=8, on peut lancer la méthode **Knn(train, test, k)** avec train = pretest.csv, test = finaltest.csv qui nous retourne la liste de prédictions finale.

On écrit cette liste dans un fichier txt avec la méthode **EcrireLabels(predictions).**

On vérifie bien avec la méthode **Check_label(nameFile, nbLines)** que le fichier est conforme pour l'évaluation.