**Rapport « IA : Apprentissage automatique : K-NN »**

**CHUVATIN Alexandra**

**TD P**

# I – Introduction

L’objectif de ce projet est de développer un algorithme d’apprentissage : le k-nn. Celui-ci est un algorithme de Machine Learning « supervisé ».

Nous disposons d’une base de données d’apprentissage constituée d’individus dont on connaît la classe. La méthode des k plus proches voisins consiste à sélectionner les k individus les plus proches en termes de distance (distance euclidienne dans notre cas). C’est en retenant la classe la plus représentée des k voisins qu’on déterminera la nature de l’individu testé.

Il est donc nécessaire d’avoir une valeur pour le paramètre k afin de répondre à ce problème. Cependant, d’autres facteurs sont à déterminer. C’est ce que nous allons voir par la suite, en plus de l’algorithme en lui-même.

# II – Plan d’action

1. Récupérer les données

Dans un premier temps, il nous était demandé de découvrir l’algorithme k-nn à l’aide du dataset Iris qui présente 4 paramètre (longueur et largeur de pétales, longueur et largeur des sépales), ainsi que le label de chaque iris. C’est sur ce premier dataset que j’ai commencé à travailler, mais ce sera sur le dataset « data.csv » obtenu plus tar que je vais baser mes explications.

Ce dernier comporte 5 variables quantitatives et une variable qui représente la classe. Dans l’optique de généraliser mon algorithme, je suis partie du constat qu’on ne connaît pas le nombre de variables à l’avance, mais qu’on sait que la classe est toujours donnée en dans la dernière colonne du dataset.

Pour lire le fichier et récupérer ses données, j’utilise la bibliothèque pandas de python qui permet la manipulation et l’analyse de données.

La lecture du fichier se fait comme suit :

def ReadFile(filename):

   df=pd.read\_csv(filename,sep=",",header=None)

   dimension=()

   dimension=df.shape

   nb\_var=dimension[1]

   nb\_indiv=dimension[0]

   var=[]

   for i in range(0,nb\_var-1):

     var.append(df.iloc[:,i])

   I=[]

   Itemp=[]

   for i in range(0,nb\_indiv):

     for v in var:

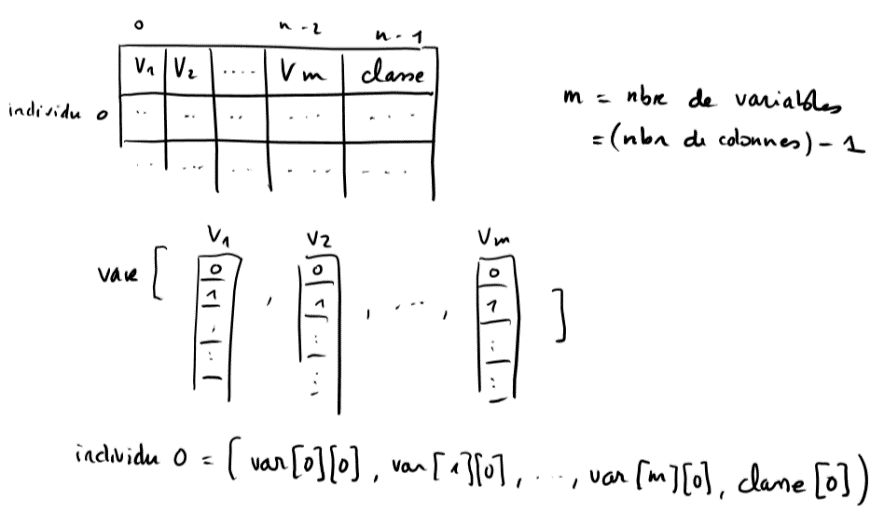
       Itemp.append(v[i])

     I.append(Itemp)

     Itemp=[]

   return I

La méthode retourne une liste I composée de tous les individus. Chaque individu est une liste composée des variables et de la classe en dernière position.

Pour me retrouver dans les différentes listes, je faisais des schémas au brouillon, et qui peuvent vous aider à comprendre mon algorithme :

1. Données de training/test

La prochaine étape était de séparer les données en données **d’apprentissage** et en données **d’évaluation**. Les données d’apprentissage servent à déterminer la classe d’un individu. Les données d’évaluation servent à tester notre algorithme sur des individus dont on connait déjà la classe. Il y a par la suite les données de validation, où ni moi, ni la machine ne connaissons la classe, qu’il faudra donc déterminer.

Pour séparer les données, il faut choisir un pourcentage : combien de données pour l’apprentissage ? et pour l’évaluation ? Ce sont des paramètres qu’il faudra tester. Prenons par exemple 80% des données pour l’apprentissage et le reste pour l’évaluation. Ma première idée était de trier les données par classe, puis dans chaque classe prélever 80% en apprentissage et 20% en évaluation. L’algorithme de tri par classe était comme suit :

def TriDonnees(data):

   return sorted(data, key=lambda x : x[len(x)-1])

Mais prélever dans chaque classe un pourcentage me paraissait compliqué à implémenter, c’est pourquoi après quelques recherches, j’ai décidé d’utiliser la méthode *train\_test\_split()* de la bibliothèque sklearn. Celle-ci permet de séparer un jeux de données en données d’apprentissage et d’évaluation en fonction du pourcentage choisi. On peut aussi régler le degré de choix (aléatoire) de distribution dans l’un ou l’autre jeu de données.

Je sépare donc mon tableau d’individus en 2 : X contiendra les données de l’individu et Y contiendra les classes.

"""

   dans X je mets les valeurs de chaque individu

   dans Y je mets les classes correspondantes

   """

   X=[]

   Y=[]

   for i in data:

     X.append(i[:len(i)-2])

     Y.append(i[len(i)-1])

Puis je créé mes données d’apprentissage et d’évaluation :

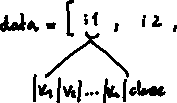
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)

test\_size = 0.2 => 20% des données servent à l’apprentissage.

Cependant, pour ma méthode qui détermine les K plus proches voisins, il me fallait une liste contenant les données d’apprentissage avec l’individu et sa classe, et non séparées en 2 listes X\_train et Y\_train.

Finalement, j’ai décidé de ne pas trier mes données et ne pas utiliser *train\_test\_split()*. Au lieu de cela, j’utilise la méthode shuffle de sklearn.utils pour mélanger mon tableau d’individus, puis je calcule combien d’individus je dois prendre pour les données d’apprentisage en fonction du pourcentage choisi (si 80% : part=0.8\*len(data)).

Où data est mon tableau d’individus de cette forme :



Pour finir, je mets les premiers 80% en apprentissage :

train = data[ :part]

et les 20% restants en évaluation :

test = data[part : ]

1. Prédiction des classes : meilleur k ?

Il faut maintenant évaluer la précision de notre algorithme k-nn grâce aux données d’évaluation. Pour ce faire, il faut aussi déterminer quel est le k qui donne le meilleur résultat.

Pour déterminer le meilleur k, nous allons comparer tous les résultats obtenus pour k allant de 1 à 30 (j’ai testé pour des k plus grands mais les valeurs n’étaient vraiment pas optimales, on diminue donc la borne sup à 30). Les résultats viennent de la comparaison de la vraie classe des individus d’évaluation avec la classe prédite grâce aux données d’apprentissage. Le résultat est en fait le pourcentage de réussite calculé en comptant combien de classe ont été prédites de manière exacte grâce à la méthode ci-dessous :

def Exact(prediction, realite):

  cpt=0

  for i in range(len(prediction)):

    if realite[i] == prediction[i]:

      cpt+=1

  cpt=cpt\*100/(len(realite)-1)

  return cpt

Il suffisait ensuite d’afficher pour chaque cas le pourcentage de réussite grâce au code suivant :

#vraies classes

   realite=[]

   for i in range(len(test)-1):

     realite.append(test[i][6])

   #classes prédites

   prediction=[]

   for k in range(1,30):

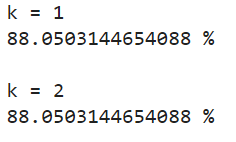
     print("k =",k)

     for i in range(len(realite)):

       prediction.append(KPlusProche(train, k, test[i]))

     print(Exact(prediction,realite),"%","\n")

     prediction=[]



Ce qui donnait une sortie console de ce type :

Cependant, pour mieux visualiser les résultats obtenus, j’ai utilisé la bibliothèque matplotlib.pyplot en affichant le pourcentage de réussite en fonction de k.

Méthode :

def Reussite\_fct\_K(realite):

  P=[]

  prediction=[]

  for k in range(1,30):

    #print("k =",k)

    for i in range(len(realite)):

      prediction.append(KPlusProche(train, k, test[i]))

    P.append(Exact(prediction,realite))

    #print(Exact(prediction,realite),"%","\n")

    prediction=[]

  K=range(1,30)

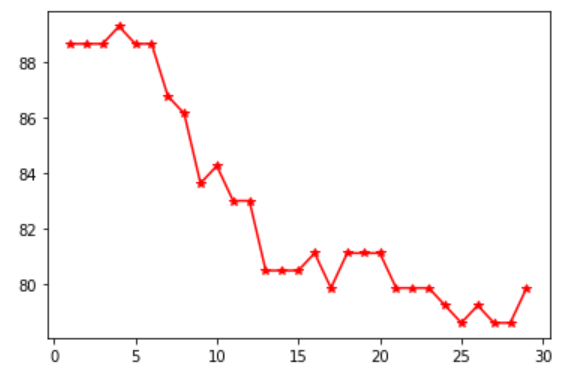
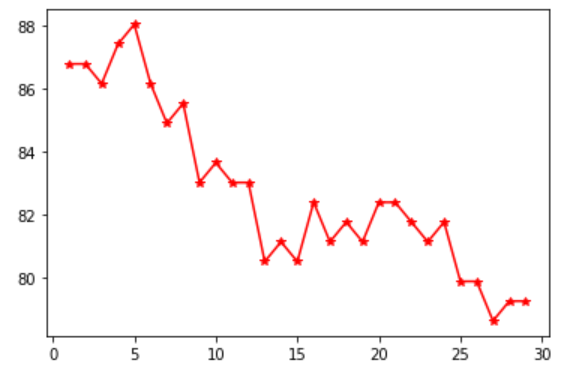
   plt.plot(K,P,"r\*-")

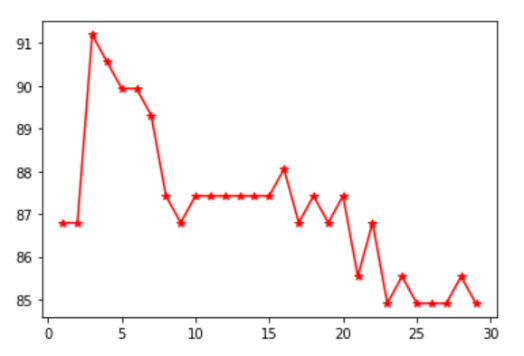
   plt.title("Taux de réussite en fonction de k")

   plt.xlabel("-- k --")

   plt.ylabel("Taux de réussite")

  plt.show()

Voici les courbes obtenues sur 3 itérations :



*Pourcentage de réussite en fonction de k sur 3 itérations*

On remarque que le meilleur k varie entre 3 et 5 (valeurs testés sur plus de 3 itérations), mais la valeur qui ressort le plus souvent est 4.

En testant de la même manière sur le nouveau fichier « preTest.csv », on retombe de la même manière sur un k tourne autour de 4.

On prendra donc **k = 4**.

1. Prédiction des classes : algorithme k-nn

Nous avons réussi à déterminer le meilleur k. Il reste maintenant à appliquer l’algorithme des k plus proches voisins :

def KPlusProche(train, k, test):

  train=sorted(train, key=lambda x: Distance(test, x))

  #Les k plus proches voisins

  top=[]

  for i in range(k):

    top.append(train[i])

  #les classes des k plus proches voisins

  labels=[]

  for i in top:

    labels.append(i[len(i)-1])

  #récuperer la classe qui apparaît le plus souvent

  cpt=0 #compte le nombre d'apparition de la classe

  index=0 #index de l'individu qui a la classe qui appraît le plus souvent

  for i in range(len(labels)):

    if labels.count(labels[i]) > cpt:

      cpt = labels.count(labels[i])

      index=i

  return labels[index]

La méthode prend en paramètre les données d’entrainement (« preTest.csv »), le k (k=4) et les données de test (les individus de « finalTest.csv »).

La première étape est de classer les données de train selon leur distance euclidienne par rapport à l’individu testé. Celle-ci est calculée grâce à la méthode Distance qui prend en paramètre l’individu testé et un individu des données d’entrainement :

def Distance(i1,i2):

  dist=0

  for index in range(0,len(i1)-2):

    dist+=(float(i1[index])-float(i2[index]))\*\*2

  return np.sqrt(float(dist))

1. Matrice de confusion

Pour bien visualiser les résultats obtenus, nous pouvons utiliser une matrice de confusion qui montre la qualité du résultat obtenu en fonction des classes initiales. Elle donne pour chaque classe le nombre d’éléments ont été bien prédits, ainsi que le nombre d’éléments qui ont été prédits par la mauvaise classe.

Pour afficher la matrice de confusion, j’ai d’abord voulu utiliser la méthode *plt\_confusion\_matrix()*, mais je n’ai pas réussi à bien l’utiliser. J’ai donc créé ma matrice de confusion à l’aide de la méthode *confusion\_matrix(),* que j’ai ensuite affiché grâce à la méthode *plt.imshow().*

* **Création de la matrice :**

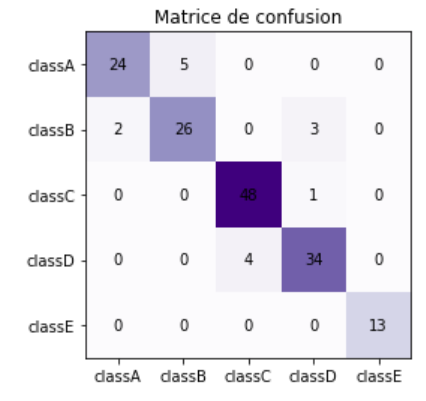
matrice=confusion\_matrix(realite, prediction)

où « realite » et « prediction » ont été calculées précédemment.

* **Affichage de la matrice :**

plt.imshow(matrice)

Enfin, après avoir ajouté le nom des classes, les valeurs dans chaque case, et avoir choisi une couleur, voici la matrice de confusion obtenue :



1. Prédiction finale

Enfin, la dernière étape était de générer le fichier de prédiction des classes du fichier *« finalTest.csv ».* Pour ce faire, nous utilisons le même code que dans la partie c) qui retourne le tableau prediction :

prediction=[]

  for i in range(len(test)):

    prediction.append(KPlusProche(train, 4, test[i]))

  Resultats(prediction)

La dernière ligne appelle la méthode *Resultats* qui est en fait la méthode qui créé le fichier de sortie au format .txt avec les classes prédites :

def Resultats(prediction):

  prediction="\n".join(prediction)

  with open("chuvatin.txt","w") as f:

    f.write(prediction)

  f.close()

NB : Pour prédire les classes, j’ai d’abord utilisé le fichier *« preTest.csv ».* J’ai ensuite voulu essayer de voir si en combinant les fichiers *« data.csv »* et *« preTest.csv »* on obtenait des résultats différents, car il y avait alors 1606 données d’apprentissage. Le fichier de sortie s’est avéré totalement identique. Je peux donc utiliser uniquement *« preTest.csv »* pour gagner en temps d’exécution.

 #LECTURE DU FICHIER

  I1=ReadFile("preTest.csv")

  I2=ReadFile("data.csv")

  I=I1+I2

  Itest=ReadFileTest("finalTest.csv")

OU

 #LECTURE DU FICHIER

  I=ReadFile("preTest.csv")

  Itest=ReadFileTest("finalTest.csv")

# III – Conclusion

Finalement, des variables qu’il fallait déterminer (k, répartition des données en apprentissage et évaluation) nous retenons :

**k = 4**

**80% — apprentissage**

**20% — évaluation**

Et le fichier de prédiction des classes est obtenu grâce au fichier *« preTest.csv ».*