Classification KNN

Goubet Victor
Graff Thomas
TD D

Ce document a été rédigé dans le but de garder une trace de ce travail. Il est donc exaustif et retrace toutes les étapes de notre démarche

Partie I

Prepatation des données

On commence par importer les librairies necessaires :

Entrée [78]:

```
import pandas as pd

from sklearn import model_selection
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing

import numpy as np
import matplotlib.pyplot as plt
```

On importe les données :

Entrée [79]:

```
data= pd.read_csv('data.csv',sep=';',header=None)
```

Ici on stock dans le vecteur X les colonnes correspondant aux classe et dans Y celles correspondant aux etiquettes

Entrée [80]:

```
X=data[data.columns[:-1]].values
Y=data[data.columns[-1]].to_numpy()

labels_Class=np.unique(Y)
```

On va maintenant normaliser nos données. Cela permettra de réduire l'impact des données erronées. Cela revient donc à diviser nos données par l'ecart type. Dans un soucis de rapidité on utilise ici le scaler fournis par sklearn.

Entrée [81]:

```
std_scale = preprocessing.StandardScaler().fit(X)
X_normalize = std_scale.transform(X)
```

On utilise le module Sklearn pour séparer nos données en un jeux d'entrainement et un autre de test.
La separation est plus efficace car il procéde à un mélange aléatoire avant de séparer les données

Entrée [82]:

```
X_train, X_test, Y_train, Y_test = model_selection.train_test_split(X_normalize, Y,test_siz
#On fixe le random state afin d'avoir toujours la même distribution aléatoire
```

Definition du Model

Entrée [83]:

```
class KNN:

    def __init__(self,k=1):
        self.k=k

    def Train(self,X_train,Y_train):
        self.X_train=X_train
        self.Y_train=Y_train

    def Distance(self,ptTest,ptTrain):
        #On commence par définir une fonction de calcul de distance. Ici on choisit la dist
        distance=0
        for k in range(len(ptTrain)):
            distance+=(ptTrain[k]-ptTest[k])**2
        return np.sqrt(distance)


    def GetLabel(self,instance):
        #On definit la fonction GetLabel permettant de prédir l'étiquette d'une instance
        distances=[self.Distance(instance,x) for x in self.X_train ]
        indexes=np.argsort(distances)[:self.k]
        Top_Neighbours=np.array([self.Y_train[i] for i in indexes])

        return max(labels_Class,key=lambda x:np.sum(Top_Neighbours==x))

    def GetLabels(self,X):
        #On récupére ici le label de chaque instance
        labels=[self.GetLabel(x) for x in X]
        return np.array(labels)

    def Precision(self,X_test,Y_test,MC):
        #On compare nos label estimés aux réel labels et on en déduit une precision
        labels=self.GetLabels(X_test)
        precision=np.sum(labels==Y_test)/len(labels)*100

        if MC:
            fig = plt.figure(figsize=(5,5))
            ax = fig.add_subplot(111)
            im = ax.imshow(confusion_matrix(Y_test,labels))
            fig.colorbar(im)
            plt.title("Matrice de confusion")
            plt.xlabel("Predicted")
            plt.ylabel("Actual")

        return precision
```

Test du model

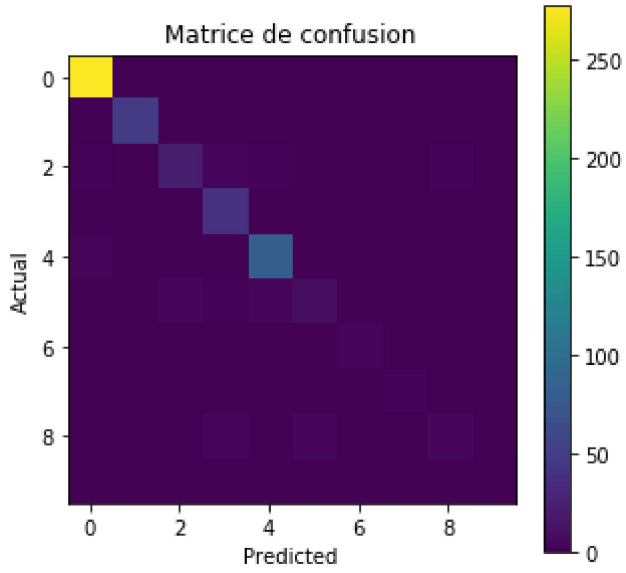
On test notre fonction avec le resultat labels utilisant k=2
On affiche le pourcentage de precision et la matrice de confusion

Entrée [84]:

```
model=KNN(2)
model.Train(X_train,Y_train)
model.Precision(X_test,Y_test,True)
```

Out[84]:

```
86.11111111111111
```



En prenant un k aléaoirement on a ici un score de 86.11

Optimisation du model

Maintenant que notre modèle est construit et qu'il nous fournit des resultats acceptables, nous pouvons essayer de régler le paramètre k afin d'augmenter notre score. Pour cela on va séparer nos données non pas en deux mais en 3 sets. Un pour l'entrainement, un pour la validation (optimisation de k) et un dernier pour le test final.

Cependant, plusieurs découpages de nos données sont possibles pour choisir notre set de validation. On va donc utiliser la méthode de **cross validation** qui permet d'explorer tout les découpages possibles pour notre set et de retourner à chaque fois son score.
Ici on va utilise le découpage des **KFolds**

Entrée [85]:

```
def CrossValidation(k,X_train,Y_train,nFolds=5):
    scoresValidation=[]

    model=KNN(k)
    X_Splits=np.array_split(X_train,nFolds)
    Y_Splits=np.array_split(Y_train,nFolds)

    for i in range(nFolds):
        print("\nCalcul du fold n°(",i+1,"/",nFolds,")..")
        X_validation=X_Splits[i]
        Y_validation=Y_Splits[i]
        X_newTrain=np.concatenate(np.delete(X_Splits,i,0))
        Y_newTrain=np.concatenate(np.delete(Y_Splits,i,0))

        model.Train(X_newTrain,Y_newTrain)
        scoresValidation.append(model.Precision(X_validation,Y_validation,False))

    return np.array(scoresValidation)
```

On peut maintenant tester notre methode pour k=2 et 5 folds

Entrée [86]:

```
scoresValidation=CrossValidation(2,X_train,Y_train)
print("\nScore sur le set de validation pour chaque fold :\n",scoresValidation)
print("\nMoyenne des scores:",scoresValidation.mean())
```

```
Calcul du fold n°( 1 / 5 )..

Calcul du fold n°( 2 / 5 )..

Calcul du fold n°( 3 / 5 )..

Calcul du fold n°( 4 / 5 )..

Calcul du fold n°( 5 / 5 )..

Score sur le set de validation pour chaque fold :
 [83.51409978 85.90021692 84.13043478 86.30434783 83.91304348]

Moyenne des scores: 84.7524285579553
```

Nous sommes maintenant prêt à tester pour differents valeures de k
On va tracer le score sur le set de validation en fonction de k.

Entrée [87]:

```
def Optimise(listeK,X_train,Y_train):
    scoresValidation=[]

    for k in listeK:
        print("============================Test k=",k,"==================================")

        accuracy=CrossValidation(k,X_train,Y_train,5).mean()
        scoresValidation.append(accuracy)
        print("\n→ Accuracy =",accuracy)
    return scoresValidation
```

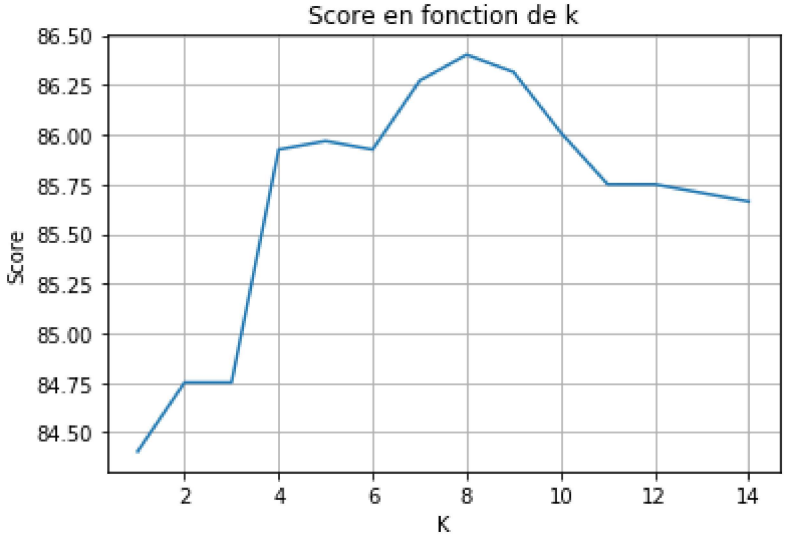
Entrée [88]:

```
listeK=range(1,15)
scoresValidation=Optimise(listeK,X_train,Y_train)
```

```
Calcul du fold n°( 4 / 5 )..

Calcul du fold n°( 5 / 5 )..

→ Accuracy = 86.31547675186269
============================Test k= 10 ==================================

Calcul du fold n°( 1 / 5 )..

Calcul du fold n°( 2 / 5 )..

Calcul du fold n°( 3 / 5 )..

Calcul du fold n°( 4 / 5 )..

Calcul du fold n°( 5 / 5 )..

→ Accuracy = 86.01131755163632
============================Test k= 11 ==================================
```

Entrée [89]:

```
plt.plot(listeK,scoresValidation)
plt.title("Score en fonction de k")
plt.ylabel("Score")
plt.xlabel("K")
plt.grid()
```



On peut voir ici que l'on obtient des scores haut pour k=7-8-9. Cependant un k pair peu mener à des votes égalitaires. On choisit donc k=9.

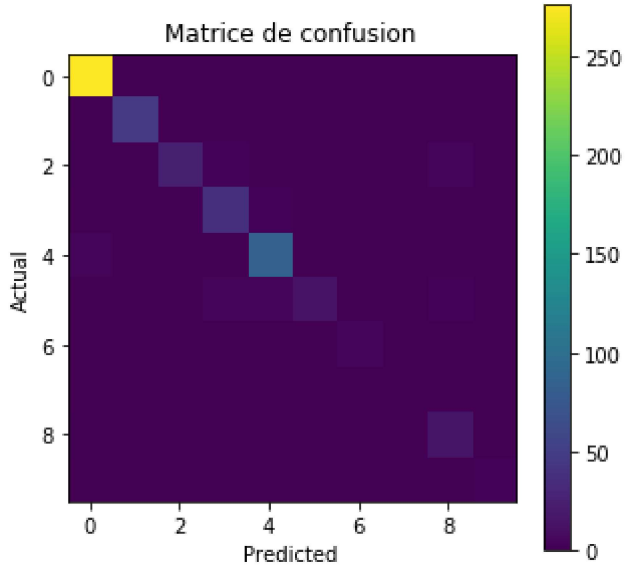
Test final

Entrée [91]:

```
model=KNN(9)
model.Train(X_train,Y_train)
model.Precision(X_test,Y_test,True)
```

Out[91]:

```
88.36805555555556
```



On obtient finalement un socre de **88.37**.

Ameliorations

Une première amélioration serait d'optimiser les calculs afins qu'ils soient plus rapides.Plusieurs techniques ont déja été imaginées comme l'utilisation d'un ACP pour réduire le nombre de dimension pour la distance ou encore utiliser des arbres de recherche pour trouver des "presques voisin"
Une deuxième amélioration non nègligeable serait d'optimiser un deuxième hyper paramètre qui est le calcul de la distance en utilisant par exemple la distance manathane. Cependant cela reviendrait à chercher la solution dans une grille (distance en ligne et k en colonne) soit environs 5x15 possibilitées (5 types de distance et on test 15 k) ce qui prendrait enormement de temps avec une version non optimisée.
On revient donc à la première améliroation..

Partie II

Nous avons maintenant un deuxième dataset à diposition. Nous avons choisit d'utiliser ce dataset afin d'agrandir nos données d'aprentissage. En effet un plus grand nombre de données d'entrainement améliorera la precision de notre model (on peut d'ailleurs le voir sur les learning curve avec sklearn).
Dans un premier temps nous allons tester notre model sur ces datas voir si tout se passe bien, puis, nous fusionnerons les deux datasets et nous l'entrainerons avec ces nouvelles données.

Test sur le nouveau dataset

on prépare nos donnèes

Entrée [99]:

```
data2= pd.read_csv('preTest.csv',sep=';',header=None) #On charge

#On sépare les etiquettes
X2=data2[data2.columns[:-1]].values
Y2=data2[data2.columns[-1]].to_numpy()

#On normalise
std_scale2 = preprocessing.StandardScaler().fit(X2)
X2_normalize = std_scale2.transform(X2)
```

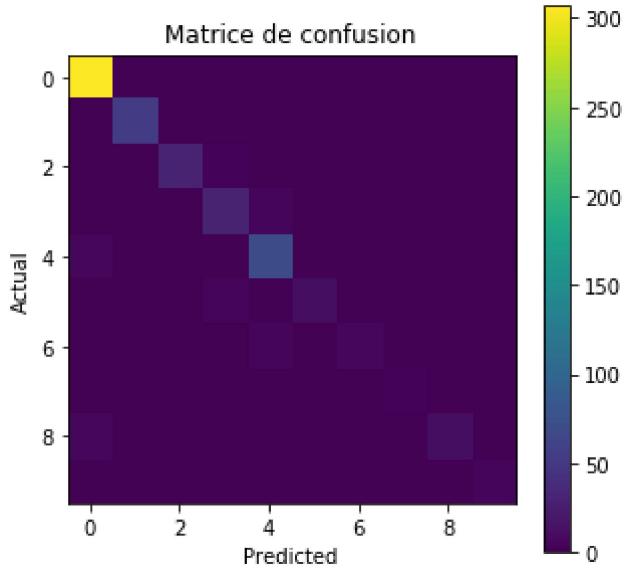
On regarde le score pour notre nouveau dataSet.

Entrée [100]:

```
model2=KNN(9)
model2.Train(X_train,Y_train)
model2.Precision(X2_normalize,Y2,True)
```

Out[100]:

88.0



On obtient ici avec le model optimisé un score de **88.0**. Tout semble donc marcher.
Un premier test a été fait en "fitant" notre model au total des données du premier dataset, cela aurait pu améliorer les perfomances car il y a ainsi plus de données de référence. Cependant cela a légérement baissé le score. Cela est peut être du à un phénomene de **double descent**. Dans le doute, nous laissons X_train et Y_train en données d'entrainement

Fusion des dataSets

Fusionnons maintenant les dataset afin d'augmente notre precision

Entrée [101]:

```
X3_normalize=np.concatenate([X_normalize,X2_normalize])
Y3=np.concatenate([Y,Y2])

X_train3, X_test3, Y_train3, Y_test3 = model_selection.train_test_split(X3_normalize, Y3,te
```

On optimise de nouveau le model

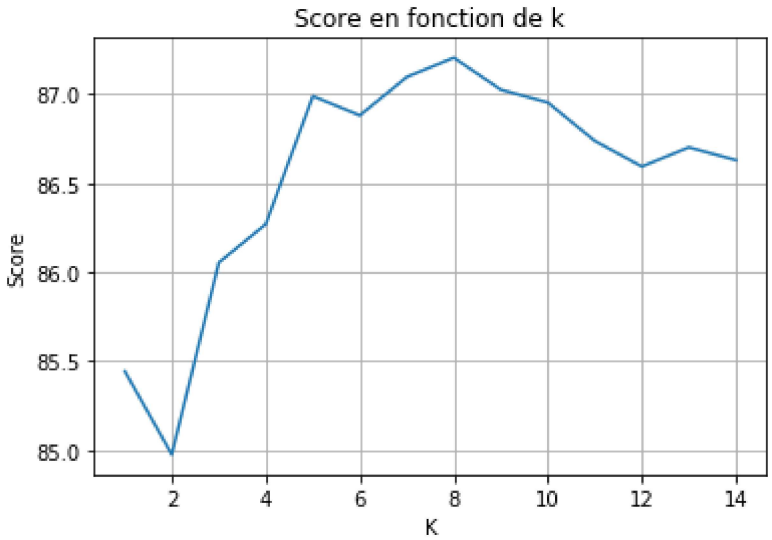
Entrée [102]:

```
scoresValidation3=Optimise(listeK,X_train3,Y_train3)
```

```
Calcul du fold n°( 2 / 5 )..

Calcul du fold n°( 3 / 5 )..

Calcul du fold n°( 4 / 5 )..

Calcul du fold n°( 5 / 5 )..

→ Accuracy = 86.70020536533072
============================Test k= 14 ==================================

Calcul du fold n°( 1 / 5 )..

Calcul du fold n°( 2 / 5 )..

Calcul du fold n°( 3 / 5 )..

Calcul du fold n°( 4 / 5 )..
```

Entrée [103]:

```
plt.plot(listeK,scoresValidation3)
plt.title("Score en fonction de k")
plt.ylabel("Score")
plt.xlabel("K")
plt.grid()
```



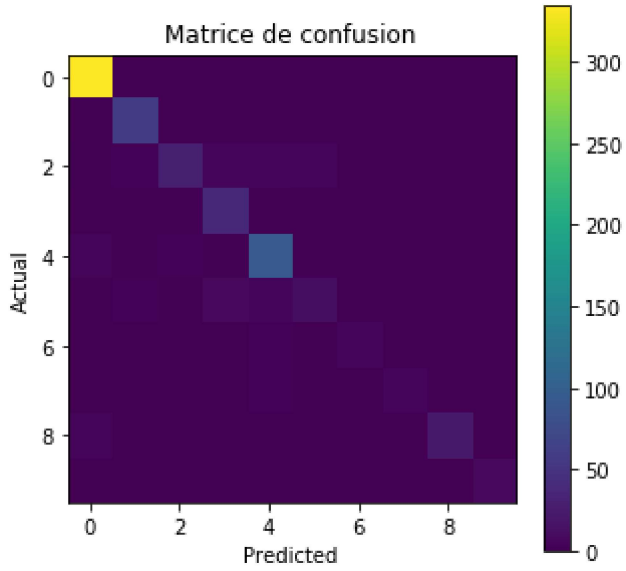
On voit ici qu'avec plus de données, le réglages k=7 semble plus optimal que k=9 (on ignore toujours k=8 car il est pair).
On va donc prendre cette valeure.

Entrée [107]:

```
modelFinal=KNN(7)
modelFinal.Train(X_train3,Y_train3)
modelFinal.Precision(X_test3,Y_test3,True)
```

Out[107]:

87.5



On obtient alors sur notre set de test un score de **87.5**

Partie III

Nous allons maintenant tester notre model sur un dataset inconnu dont on ne connait pas les etiquettes.
On va commencer par importer les données:

Entrée [108]:

```
datafinal= pd.read_csv('finalTest.csv',sep=';',header=None)
Xfinal=datafinal.values

std_scaleFinal = preprocessing.StandardScaler().fit(Xfinal)
Xfinal_normalize = std_scaleFinal.transform(Xfinal)
```

Enregistrement des labels

On enregistre les labels dans un fichier texte

Entrée [109]:

```
def Save(labels):
    with open("GOUBET_GRAFF.txt",'w') as f:
        for label in labels:
            f.write(label+"\n")
```

Entrée [110]:

```
labels=modelFinal.GetLabels(Xfinal_normalize)
Save(labels)
```

Comparaison avec le model de Sklearn

Optimisation de k

Entrée [111]:

```
from sklearn.model_selection  import validation_curve
from sklearn.neighbors import KNeighborsClassifier
```

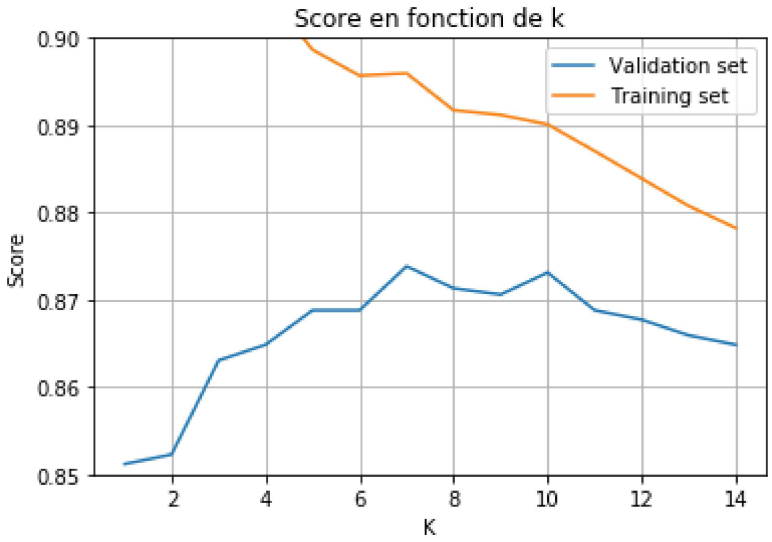
On utilise ici le model fournit par le package et l'optimisation de k par cross validation

Entrée [112]:

```
model=KNeighborsClassifier()
ScoreTrain,ScoreValidation=validation_curve(model,X_train3,Y_train3,'n_neighbors',listeK,cv

ScoreValidation_Mean=ScoreValidation.mean(axis=1)
ScoreTrain_Mean=ScoreTrain.mean(axis=1)
```

Entrée [113]:

```
plt.plot(listeK,ScoreValidation_Mean,label='Validation set')
plt.plot(listeK,ScoreTrain_Mean,label='Training set')
plt.legend()
plt.title("Score en fonction de k")
plt.ylabel("Score")
plt.xlabel("K")
plt.ylim(0.85,0.9)
plt.grid()
plt.show()
```



- On a également tracé ici la courbe de score pour le training set. Cela nous permet de reperer les zones de suraprentissages (typiquement pour k< 4-5 ici).
- On obtient donc ici k=9

GridSearch

On va ici utiliser la méthode grid search qui procède toujours par cross validation mais qui va optimiser tout les hypers parmètres (ici k et distances).

Entrée [114]:

```
from sklearn.model_selection import GridSearchCV
```

Entrée [115]:

```
parametres={'n_neighbors':listeK,'metric':['euclidean','manhattan','minkowski']}

grid_params = [
    {'n_neighbors': listeK, 'metric': ['euclidean', 'minkowski','chebyshev']},
    {'n_neighbors': listeK, 'metric': ['mahalanobis', 'seuclidean'],
     'metric_params': [{'V': np.cov(X_train.T)}]}
]

grid=GridSearchCV(KNeighborsClassifier(),parametres,cv=5)
grid.fit(X_train3,Y_train3)
print(grid.best_params_)
```

```
{'metric': 'euclidean', 'n_neighbors': 7}
```

On s'aperçoit qu'ici la distance euclidienne est la meilleure et que k=7 est la meilleure option. On trouve donc le même k trouvé plus haut.
On a par ailleurs bien fait de ne pas utiliser un distance avec poids comme la distance de minkoswki qui pourtant parait plus discriminante envers les points isolés et donc semblerait réduire le bruit mais qui ici apporte un moins bon score.

Entrée [116]:

```
model=grid.best_estimator_
model.score(X_test3,Y_test3)
```

Out[116]:

```
0.875
```

On obtient avec ce model une precision de **87.5** sur le set de test qui est égal au score trouvé plus haut.
On peut donc conclure que notre model est valable.