SOUFFLET Thomas
VARACCA Maxime

Groupe i

RAPPORT KNN

Ce compte rendu retrace le raisonnement et le travail effectués pour développer notre algorithme KNN. Il exposera comment nous avons sélectionné la valeur optimale de k, quels tests nous avons menés pour maximiser notre précision, quelles modifications ont été adoptées et celles qui ne l'ont pas été.

La majeure partie des tests ont été effectués sur le jeu de données "data.txt". Nous avons mélangé de manière aléatoire l'intégralité du jeu de données, puis l'avons divisé en deux parties distinctes: l'une pour l'apprentissage et l'autre pour le test.
Par la suite, nous avons finalisé notre programme en utilisant “data.txt” comme jeu d'entraînement et “preTest.txt” comme jeu de test.

Choix de la répartition test/apprentissage

Pour rendre notre algorithme le plus efficace possible, nous avons divisé notre jeu de données "data.txt" en deux parties - une pour l'entraînement et l'autre pour les tests. Nous avons testé différentes proportions et avons constaté qu'utiliser 80% des données pour l'entraînement et 20% pour les tests était le plus précis. Cela nous a donné environ 800 données d'entraînement et 200 données de test, ce qui était suffisant pour produire des résultats fiables pour le "finalTest.txt".

```
#Mélange les données
        df = df.sample(frac=1).reset_index(drop=True)

        #Prend 80% des données pour l'entrainement
        df_train = df.iloc[:int(len(df)*0.8)]

        #Prend 20% des données pour le test
        df_try = df.iloc[int(len(df)*0.8):]
```

Choix du k

Dans le but de trouver le meilleur k, nous avons ajouté une boucle dans notre algorithme qui explore tous les k entre 1 et 10. À chaque k testé, l'algorithme a affiché la précision obtenue ainsi que la matrice de confusion correspondante.
Suite à cela, nous avons remarqué que le k optimal était 5. De plus, nous avons constaté que les valeurs impaires de k fournissaient une précision supérieure à celle des valeurs paires, un résultat qui peut être utile pour l'optimisation future de l'algorithme, en forçant par exemple l'utilisateur de choisir un k impair.

```
k =  4
Précision :  0.9704433497536946
Matrice de confusion :
[[168.   0.   0.   4.]
 [  0.   9.   0.   0.]
 [  0.   0.  11.   0.]
 [  0.   0.   2.   9.]]
k =  5
Précision :  0.9901477832512315
Matrice de confusion :
[[164.   0.   0.   2.]
 [  0.   9.   0.   0.]
 [  0.   0.  14.   0.]
 [  0.   0.   0.  14.]]
```

```
k =  2
Précision :  0.9704433497536946
Matrice de confusion :
[[158.   0.   0.   4.]
 [  0.  11.   0.   0.]
 [  0.   0.  17.   1.]
 [  0.   0.   1.  11.]]
k =  3
Précision :  0.9852216748768473
Matrice de confusion :
[[168.   0.   0.   2.]
 [  0.   9.   0.   0.]
 [  0.   0.  13.   0.]
 [  0.   0.   1.  10.]]
```

Test de normalisation

Ensuite, nous avons tenté de normaliser nos données afin de vérifier si cela améliorerait les résultats. Cependant, après avoir effectué ces tests, nous avons observé que la normalisation diminuait la précision de notre algorithme. Nous avons donc abandonné cette idée d'amélioration.

Voici la fonction que nous avions créé pour tester la normalisation :

```
#Calcule la distance normalisée entre le point de test et un point
d'entrainement de coordonnées (a,b,c,d,e,f,g)
def distanceNormalisee(dataTest,a,b,c,d,e,f,g):
    pointa= np.array((a,b,c,d,e,f,g))
    return
(np.linalg.norm(dataTest-pointa)-np.mean(dataTest-pointa))/np.std(dataTe
st-pointa)
```

Précision pour k = 5 avec normalisation :

```
Précision :  0.7934782608695652
Matrice de confusion :
[[760.  40.  24.  22.]
 [  0.   0.  17.   0.]
 [  0.   0.  39.  54.]
 [ 50.   2.   0.   4.]]
```