Classification Challenge : Algorithme KNN

Ce rapport vise à expliquer comment a été implémenté l'algorithme d'apprentissage supervisé des K plus proches voisins.

Normalisation des données :

Pour avoir un traitement optimal de notre dataset, il est nécessaire de centrer et de réduire l'ensemble des données. En effet l'algorithme du KNN repose sur un calcul de la distance entre deux individus. Certaines variables peuvent ne pas avoir le même ordre de grandeur ce qui pourrait leur donner un poids trop significatif en cas d'écart important. Nos données seront standardisées après ce traitement.

Pour cela, on utilise cette formule sur l'ensemble des variables :

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Avec :

$\hat{x}$: valeur centrée réduite

X : valeur à centrer réduire

μ : moyenne de la série

σ : écart-type de la série

Implémentation en Python :

```
#critere de précision à prendre en compte
def normalisation(data):
  data_norm = data.copy()
  for i in range(0,len(data_norm.columns)-1):
    moy,ecart = mean(data_norm.iloc[:,i]),std(data_norm.iloc[:,i])
    for j in range(len(data_norm)):
      data_norm.iloc[j,i]=round((data_norm.iloc[j,i]-moy)/ecart,2)
  return data_norm
```

Séparation des données :

Pour aboutir à un algorithme le plus général possible et éviter le surentrainement, j'ai décidé de séparer mon dataset. J'ai d'abord combiné l'ensemble des exemples disponibles dans un dataframe « data ». Je le sépare ensuite en 3 catégories : les données d'entrainement, de test et de validation. Les données d'entrainement permettront de déterminer les paramètres. Ensuite je les teste sur les données de test. Enfin je valide définitivement mon modèle avec les données de validation.

Implémentation en Python :

```
data_final = normalisation(data_final)
data=normalisation(data.sample(frac=1).reset_index(drop=True))
data_train=normalisation(data.sample(frac=1).reset_index(drop=True)[:1000])
data_test=normalisation(data.sample(frac=0.5).reset_index(drop=True)[1000:1300])
data_valid=normalisation(data.sample(frac=1).reset_index(drop=True)[1300:])
```

Calcul de la distance :

Maintenant que nos données ont été prétraitées, nous pouvons commencer à rentrer dans le cœur de l'algorithme KNN. Il s'articule autour d'un calcul de distance entre l'individu à tester et le reste des données. Il existe plusieurs manière de calculer la distance. Après plusieurs tests j'ai opté pour la distance euclidienne au dépend de la distance de Manhattan par exemple. Ainsi on a :

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Implémentation en Python :

```
def distance_euclidienne(indData,indTest):
  dist_ind=0
  for j in range(len(indTest)-1):
    dist_ind+=sqrt((indTest[j]-indData[j])**2)
  return dist_ind
```

Avec :

indData : donnée du dataset

indTest : donnée à évaluer

L'algorithme KNN (ou K-nearest neighbours)

Cette algorithme de classification repose sur un principe. Les éléments d'une même classe ont des caractéristiques proches. Ainsi si deux individus se ressemblent (i.e. distance faible), ils appartiennent à la même classe. Le paramètre K quant à lui détermine le nombre de voisins que l'on prend en compte dans le choix de notre assignation. Nous devons créer une liste des K-voisins les plus proches de notre individu à tester en se basant sur leur distance euclidienne.

Implémentation en Python :

```
def knn(indTest,data,k):
  data["distance"]=[distance_euclidienne(data.iloc[i,:],indTest) for i in range(len(data))]
  return data.sort_values(by="distance")[:k]
```

On crée une colonne « distance » à notre dataset regroupant l'ensemble des distances euclidiennes des données par rapport à indTest. On retourne alors la liste des K premiers voisins.

Prédiction de la classe de l'individu à évaluer

Une fois la liste des K-voisins les plus proches du dataset établis, il nous reste à sélectionner la classe dominante. Si K=3, on prendra la classe la plus présente dans les 3 voisins les plus proches. Si aucune classe n'apparait plus de fois, on prendra la classe de l'individu le plus proche.

Implémentation en Python :

```
def prediction_classification(voisins_proches):
  return max(set(voisins_proches["Class"].tolist()),key=voisins_proches["Class"].tolist().count)
```

On retourne la valeur max du set des classes des K-voisins avec pour clé leur nombre d'apparition.

Boucle Final

Enfin, il ne reste plus qu'à réaliser l'opération pour chaque individu et comparer le résultat avec la valeur réelle pendant la période de test ou écrire le résultat dans un fichier texte en phase de prédiction.

Implémentation en Python :

```
def boucle_final(data,k):
  prediction=[]
  for i in range(len(data)):
    indTest=data.iloc[i,:]
    estim= prediction_classification(knn(indTest,data,k))
    prediction.append(estim)
  return prediction
```

Cette boucle crée une liste de nos résultats pendant la phase de test.

```
def verif(matrix,data):
  result_class=[]
  for i in range(len(matrix)):
    result_class.append((matrix[i][i])/(data.iloc[:,index_sol].value_counts()[i])*100)
    print(f"Ratio de la classe {data.iloc[:,index_sol].value_counts().index.tolist()[i]}
          : {(matrix[i][i])/(data.iloc[:,index_sol].value_counts()[i])*100}%")
  print(f"Ratio global : {round(sum(result_class)/len(result_class),3)}%")
```

Cette dernière fonction permet de calculer le rapport de réussite/échec de prédiction ainsi que la précision globale de notre algorithme.

```
def boucle_finalv2(dataTest,dataLabel,k):
  fichier = open("debes_sample.txt","a")
  for i in range(len(dataTest)):
    indTest=dataTest.iloc[i,:]
    estim= prediction_classification(knn(indTest,dataLabel,k))
    result=str(estim)+"\n"
    fichier.write(result)
  fichier.close()
```

Cette fonction écrit dans le fichier « debes_sample.txt » les prédictions de l'algorithme pour le set dataTest. Le set dataLabel sont nos données de référence avec des individus labellisés d'une classe.

Affichage des résultats/Main :

Pour pouvoir comparer nos résultats et évaluer notre algorithme.

Implémentation en Python :

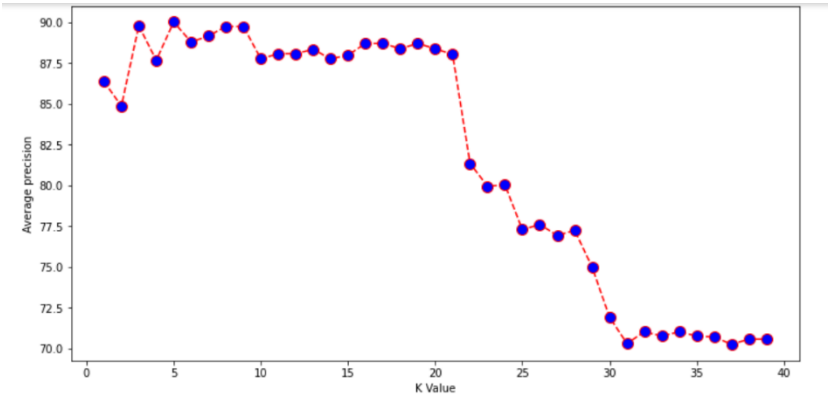
```
k=5
#Valeurs d'entrainement
prediction=boucle_final(data_train,k)
ground_truth=data_train.iloc[:,index_sol]
labels = data_train.iloc[:,index_sol].value_counts().index.tolist()
matrix = confusion_matrix(ground_truth,prediction,labels)
print(matrix)
verif(matrix,data_train)
```

```
[[202   1   3   1   0]
 [  6 152   1   3   0]
 [  4   1  95  23   0]
 [  1   3  15 101   0]
 [  0   0   0   0  38]]
Ratio de la classe classC : 97.58454106280193%
Ratio de la classe classB : 93.82716049382715%
Ratio de la classe classD : 77.23577235772358%
Ratio de la classe classA : 84.16666666666667%
Ratio de la classe classE : 100.0%
Ratio global : 90.563%
```

J'ai utilisé la fonction « confusion_matrix » de la librairie « sklearn.metrics » pour afficher la matrice de confusion.

Choix du paramètre K

Le nombre de paramètre K est l'un des paramètres que nous devons nous-mêmes définir. Pour m'aider dans cette décision, j'ai réalisé des tests avec différentes valeurs. Je me suis aussi aidé du graphique de réussite de l'algorithme en fonction de K. On voit clairement que les valeurs 3 et 5 sont les meilleures options. Il y aussi une cassure nette à K=22 où l'algorithme perd 8 points de précision. Cela montre les limites de nos classes. Au-delà des 22 voisins, il est beaucoup plus probable d'être sur le « territoire » de 2 classes.



Implémentation en Python :

```
error=[]
for k in range(1,40):
  prediction=boucle_final(data_train,k)
  ground_truth=data_train.iloc[:,index_sol]
  labels = data_train.iloc[:,index_sol].value_counts().index.tolist()
  matrix = confusion_matrix(ground_truth,prediction,labels)
  precision=[]
  for i in range(len(matrix)):
    precision.append((matrix[i][i])/(data_train.iloc[:,index_sol].value_counts()[i])*100)
  error.append(round(sum(precision)/len(precision),3))

plt.figure(figsize=(12, 6))
plt.plot(range(1, 40), error, color='red', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=10)
plt.title('Average precision depending on K value')
plt.xlabel('K Value')
plt.ylabel('Average precision')
```

Pour chaque valeur de k, on calcule la précision sur notre dataset. La précision sera alors tracée grâce à la librairie matplotlyb.

Test et Pistes d'amélioration :

Lors de la réalisation de ce projet, j'ai remarqué que malgré son apparente simplicité l'algorithme KNN possédait de nombreuses caractéristiques modifiables.

J'ai pris le parti de mettre sur un pied d'égalité toutes les variables en les centrant et réduisant. Cependant rien ne nous indique qu'elles ont le même impact. Une caractéristique pourrait être déterminante comparé aux autres. Pour vérifier cela j'ai essayé de modifier le calcul de ma distance. J'omettais certaines variables dans celui-ci. En enlevant la cinquième variable j'observais une certaine amélioration : + 0.4%. Malheureusement je n'ai pas réussi à généraliser ce résultat sur l'ensemble des données.

Dans un second temps j'ai remarqué avec ma matrice de confusion que les éléments de la Classe A avaient une forte tendance à être confondus pour des Classe D. Ces deux classes avaient les plus mauvais résultats. Pour contrer cela, j'ai rajouté à ma fonction de prédiction une condition. Si la prédiction est de classe D (possiblement une erreur de classe A), je recalcule pour un k différent. Cela n'augmentait pas énormément les résultats et rajoutaient un temps conséquent de calcul.

L'ensemble des codes évoqués se trouve en annexe de mon Google Colab.