# Data Mining : K-Nearest Neighbors

# TP N : 2

## *Objectif*

KNN (K-Nearest Neighbor) est un algorithme de classification supervisé simple que nous pouvons utiliser pour attribuer une classe à un nouvel enregistrement. Il peut également être utilisé pour la régression. KNN ne fait aucune hypothèse sur la distribution des données, il est donc non paramétrique. Il conserve toutes les données d'apprentissage pour faire des prédictions futures en calculant la similitude entre un échantillon d'entrée et chaque instance d'apprentissage.
L'objectif de ce TP est d'appliquer l'algorithme KNN sur l'ensemble de données **Iris** à l'aide de la bibliothèque scikit-learn. L'ensemble de données sur l'iris contient 50 échantillons pour chaque espèce différente de fleur d'iris (total de 150). Pour chaque échantillon, nous avons la longueur des sépales, la largeur et la longueur et la largeur des pétales et un nom d'espèce (classe/étiquette).

Ce TP couvre les aspects suivants :

1. Comment utiliser les ensembles de données de la librairie Scikit-Learn.
2. Comment diviser l'ensemble de données en un ensemble d'apprentissage et de test à l'aide de Scikit-Learn.
3. Comment utiliser KNN à l'aide de Scikit-Learn.
4. Comment choisir la valeur **K** qui représente le nombre de voisins.
5. Comment évaluer la prédiction de KNN avec Scikit-Learn.

## *I. Importez les librairies python*

Pour réaliser ce TP, on a besoin d'importer plusieurs librairies à savoir :

- Importez **sklearn.datasets** pour récupérer le contenu de l'ensemble de données **Iris**
  → `from sklearn.datasets import load_iris`
- Importez la bibliothèque **matplotlib** pour tracer et visualiser des données sous formes de graphiques.
  → `import matplotlib.pyplot as plt`
- Importez **train_test_split** à partir de la bibliothèque **sklearn.model_selection** pour diviser l'ensemble de données en un ensemble d'apprentissage et de test.
  → `from sklearn.model_selection import train_test_split`
- Importez **KNeighborsClassifier** à partir de la bibliothèque **sklearn.neighbors** pour construire l'arbre de décision.
  → `from sklearn.neighbors import KNeighborsClassifier`
- Importez **classification_report, confusion_matrix** à partir de la bibliothèque **sklearn.metrics** pour assurer l'évaluation.

→ from sklearn.metrics import classification_report,confusion_matrix

## *II. Obtenez les données*

Le package **sklearn.datasets** intègre de petits ensembles de données appelé « Toy datasets ». Ce package propose également des aides pour récupérer des ensembles de données plus volumineux couramment utilisés par la communauté de l'apprentissage automatique pour comparer les algorithmes sur les données provenant du « monde réel ».

scikit-learn dispose quelques petits ensembles de données standard qui ne nécessitent de télécharger aucun fichier à partir d'un site Web externe. Ils peuvent être chargés à l'aide des fonctions suivantes :

| Pramètres | Description |
|---|---|
| load_iris(*[, return_X_y, as_frame]) | Charger et renvoyer l'ensemble de données **Iris** (classification). |
| load_diabetes(*[, return_X_y, as_frame]) | Charger et renvoyer l'ensemble de données sur le **diabète** (régression). |
| load_digits(*[, n_class, return_X_y, as_frame]) | Charger et renvoyer le jeu de données de chiffres (classification). |
| load_linnerud(*[, return_X_y, as_frame]) | Charger et renvoyer l'ensemble de données d'exercice physique Linnerud. |
| load_breast_cancer(*[, return_X_y, as_frame]) | Charger et renvoyer l'ensemble de données sur le cancer du sein au Wisconsin (classification). |

### *Charger le jeu de données Iris et vérifier les fonctionnalités*

1. Importez la fonction **load_iris** du module d'ensembles de données **scikit-learn** et créez un objet **Bunch** iris (bunch est un type d'objet spécial de scikit-learn pour stocker des ensembles de données et ses attributs).

```
#Importez la fonction load_iris du module d'ensembles de données
from sklearn.datasets import load_iris

#créez un objet Bunch contenant l'ensemble de données iris et ses attributs
iris = load_iris()
```

2. Explorez le type de l'objet iris en utilisant la fonction **type()**.
3. Afficher les dimensions de l'ensemble de données **iris**.

```
#Dimensions de iris
print(iris.data.shape)
```

4. Affichez le contenu de l'ensemble de données iris.

```
#imprimer les données de l'iris
print(iris.data)
```

5. Affichez le nom des attributs.

```
#Noms des 4 attributs
print(iris.feature_names)
```

6. Affichez le nom des classes.

```
# 3 classes de cible
print(iris.target_names)
```

7. Affichez le contenu de la colonne prédictive.

```
# Entiers représentatnt les classes: 0=setosa, 1=versicolor, 2=virginica
print(iris.target)
```

## *III. Ensemble d'apprentissage et de test*

L'entraînement et le test sur les mêmes données n'est pas une approche optimale, nous divisons donc les données en deux parties : ensemble d'apprentissage et ensemble de test. Nous utilisons la fonction **'train_test_split'** pour diviser les données. Le paramètre facultatif "**test-size**" détermine le pourcentage de fractionnement. Le paramètre **'random_state'** divise les données de la même manière à chaque exécution.

Tout d'abord, il faut préciser les colonnes qui représentent les attributs et celle qui représente la classe.

```
from sklearn.model_selection import train_test_split

#Identifier les attributs
X=iris.data

#Identifier les classes
y=iris.target

#Diviser les données en ensemble d'apprentissage et de test (80:20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4)
```

```
#Dimension des instances d'apprentissage et de test
print(X_train.shape)
print(X_test.shape)
```

```
#Dimension des instances d'apprentissage et de test
print(X_train.shape)
print(X_test.shape)
```

## *IV. Utilisation et évaluation de KNN*

- Scikit-learn est soigneusement organisé en modules, afin que nous puissions importer facilement les classes pertinentes.

  → Importez la classe « **KNeighborsClassifer** » du module « **neighbours** » et instanciez l'estimateur (« estimateur » est le terme de scikit-learn pour un modèle). Nous appelons modèle un estimateur car leur rôle principal est d'estimer des quantités inconnues.

- Dans notre exemple, nous créons une instance de la classe « **KNeighborsClassifer** », en d'autres termes, nous avons créé un objet appelé « **knn** » qui sait comment faire la classification KNN une fois que nous avons fourni les données. Le paramètre 'n_neighbors' est le paramètre qui représente le nombre de voisins **(k)**.

- La méthode **'fit'** est utilisée pour entraîner le modèle sur les données d'entraînement (X_train,y_train) et la méthode **'predict'** pour effectuer les tests sur les données de test (X_test).

- Le choix de la valeur optimale de **K** est essentiel, nous ajustons et testons donc le modèle pour différentes valeurs de K (de 1 à 25) à l'aide d'une boucle for et enregistrons la précision des tests de KNN dans une variable (scores).

```
#Importez la classe KNeighborsClassifier à partir de sklearn
from sklearn.neighbors import KNeighborsClassifier

#Importez metrics pour vérifier l'accuracy
from sklearn import metrics

#Exécuter l'agorithme depuis k=1 jusqu'à 25 et enregistrer les taux d'accuracy
k_range = range(1,26)
scores = {}
scores_list = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred=knn.predict(X_test)
    scores[k]=metrics.accuracy_score(y_test,y_pred)
    scores_list.append(metrics.accuracy_score(y_test,y_pred))
```

## *V. Prédiction & Evaluation*

### *Afficher les résultats d'évaluation*

Le module **sklearn.metrics** implémente plusieurs fonctions évaluant l'erreur de prédiction à des fins spécifiques. Ces métriques sont catégorisées selon la tâche Data mining réalisé, à savoir : les métriques de classification, les métriques de régression et les métriques de clustering.

Nous allons maintenant voir à quel point notre algorithme est précis. La bibliothèque Scikit-Learn contient les méthodes **classification_report()** et **confusion_matrix()** qui peuvent être utilisées pour calculer ces métriques.

Pour plus d'information sur le module **sklearn.metrics :** https://scikit-learn.org/stable/modules/model_evaluation.html

```
#Importer les bibliothèques nécessaires
from sklearn.metrics import classification_report,confusion_matrix

#Afficher le taux de pertinence obtenu pour chaque valeur de K
print (scores_list)

#Afficher la matrice de confusion
print(confusion_matrix(y_test,y_pred))

#Afficher le rapport des résultats obtenu
print(classification_report(y_test,y_pred))
```

## ***Visualiser la relation entre les valeurs de K et la précision :***

Afin de tracer la relation entre les valeurs de K et la précision de test correspondante, nous utilisons l'interface **pyplot** de la bibliothèque **matplotlib**.

**Pyplot** est une collection de fonctions qui font fonctionner matplotlib comme MATLAB. Chaque fonction **Pyplot** apporte des modifications à une figure : crée une figure, crée une zone de traçage dans une figure, trace des lignes dans une zone de traçage, décore le tracé avec des étiquettes, etc. Il existe différents tracés qui peuvent être utilisés dans **Pyplot** : tracé linéaire, contour, histogramme, nuage de points, tracé 3D, etc.

La fonction **plot()** du module **pyplot** de la bibliothèque **matplotlib** est utilisée pour créer un tracé de regroupement hexagonal 2D des points x, y.

```
#importer la bibliothèque matplotlib
import matplotlib.pyplot as plt

#Tracer la relation entre K et les taux de pertinence
plt.plot(k_range,scores_list)

#Ajouter un libellé aux axes X et Y
plt.xlabel("Valeur de K")
plt.ylabel("Taux d'accuracy")
```

Selon la figure obtenue, il y a une augmentation et une diminution de la précision et c'est assez typique lors de l'examen de la complexité du modèle avec la précision. En général, lorsque la valeur de K augmente, il semble y avoir une augmentation de la précision et à nouveau elle diminue.

En général, la précision de l'apprentissage augmente au fur et à mesure que la complexité du modèle augmente. Pour **KNN**, la complexité du modèle est déterminée par la valeur de K. Une valeur de K plus élevée conduit à une limite de décision plus lisse (modèle moins complexe). Un K très petit conduit à un modèle plus complexe.

Nous obtenons la précision de test maximale lorsque le modèle a le bon niveau de complexité. Pour notre classificateur, nous pouvons voir que pour une valeur K comprise entre **3** et **19**, la précision de notre modèle est de **96,6%.**

## *VI. Modèle final*

Pour notre modèle final, nous pouvons choisir une valeur optimale de **K** égale à **5** (qui se situe entre 3 et 19) et entrainer le modèle avec toutes les données disponibles. Et ce sera notre modèle final qui est prêt à faire des prédictions.

Nous souhaitons, à présent faire appliquer le classificateur sur deux nouveaux enregistrements afin savoir quelle classe sera prédite pour chaque instance.

```
#Appliquer l'algorithme avec K=5
knn=KNeighborsClassifier(n_neighbors=5)
knn.fit(X,y)

#Affecter un libellé à chaque classe
classes={0:'setosa', 1:'versicolor', 2:'virginica'}

#faire des prédictions sur certaines données
#prédire pour les deux observations aléatoires ci-dessous
x_new=[[3,4,5,2],[5,4,2,2]]
y_predict = knn.predict(x_new)

#Afficher le résultat obtenu pour chaque cas
print(classes[y_predict[0]])
print(classes[y_predict[1]])
```