Compte rendu TP1

Matière : Diagnostic par apprentissage

ING 2 INSTRUMENTATION

Elaboré par :

MHADHEBI Zied
Ait Nouri Rayane

Exercice 1 : Construction du jeu d'entrainement et de test

1. Importer le jeu de données de fleurs d'iris en utilisant :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target
```

Le jeu de données d'iris est une base standard utilisée pour tester des algorithmes de machine learning. Elle contient des informations sur trois types de fleurs avec des caractéristiques mesurées (longueur et largeur des sépales et pétales).

2. Commande : print(X.shape)

   La commande X.shape renvoie le nombre d'exemples (lignes) et le nombre de caractéristiques (colonnes).

   (150, 4) Signifie qu'il y a 150 exemples et 4 caractéristiques.

```
In [2]: runfile('C:/Users/e12311954/Desktop/DPA/
ex1.py', wdir='C:/Users/e12311954/Desktop/DPA')
(150, 4)
```

3.

Le vecteur y contient les labels des classes. La fonction np.unique permet d'identifier les labels distincts :

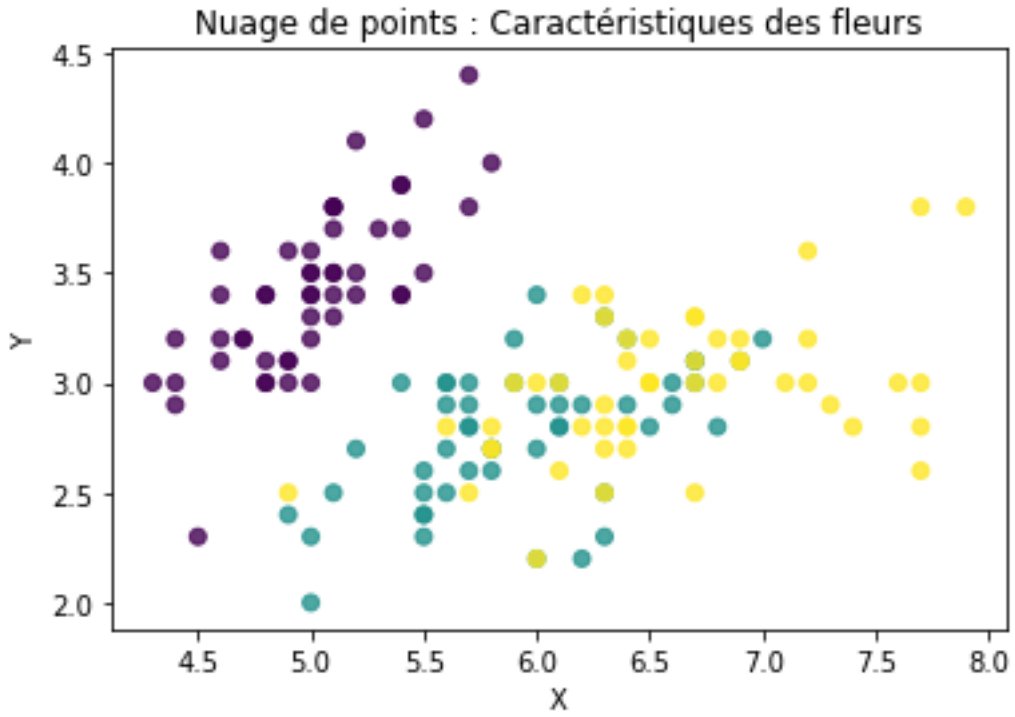
```
ex1.py', wdir='C:/Users/e12311954/Desktop/DPA')
(150, 4)
[0 1 2]
```

On remarque q'il y a 3 labels (0, 1, 2), représentant les trois espaces de fleurs.

4. Les classes identifiées sont équivalentes au nombre de labels distincts, soit trois (Setosa, Versicolor, Virginica).

5. Affichage du nuage de points

```
plt.scatter(X[:, 0], X[:, 1], c=y, alpha=0.8)
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Nuage de points : Caractéristiques des fleurs")
plt.show()
```



Interprétation :

Chaque point représente une fleur, et sa couleur indique sa classe. (Trois classes)

On peut distinguer trois groupes et il existe 150 fleurs.

6. Division du jeu de données en :

a) Prenant t=0.5 :

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
print('Train set :', X_train.shape)
print('Test set :', X_test.shape)
```

Résultat :

```
Train set : (75, 4)
Test set : (75, 4)
```

Interprétation :

Avec t=0.5, la moitié des données est pour l'entraînement et l'autre moitié pour le test (comme on peut remarquer 75 et 75)

b) Prenant t=0.2:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print('Train set :', X_train.shape)
print('Test set :', X_test.shape)
```

résultat :

```
Train set : (120, 4)
Test set : (30, 4)
```

Interprétation :

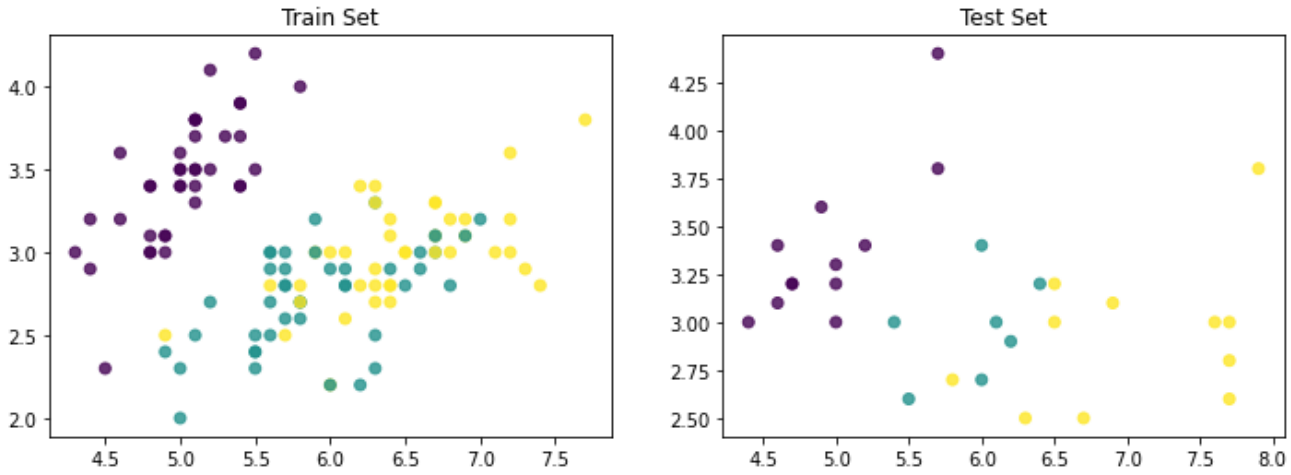
Avec t=0.2, 80% des données sont utilisées pour l'entrainement et 20% pour le test.

c)

Le paramètre t contrôle la proportion des données réservées au test.

7. On souhaite prendre 80% des données pour l'entrainement et le reste pour le test.
   a) Visualisation du nuage de points en utilisant la fonction scatter

```
plt.figure(figsize=(12, 4))
plt.subplot(121)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, alpha=0.8)
plt.title("Train Set")

plt.subplot(122)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, alpha=0.8)
plt.title("Test Set")
plt.show()
```



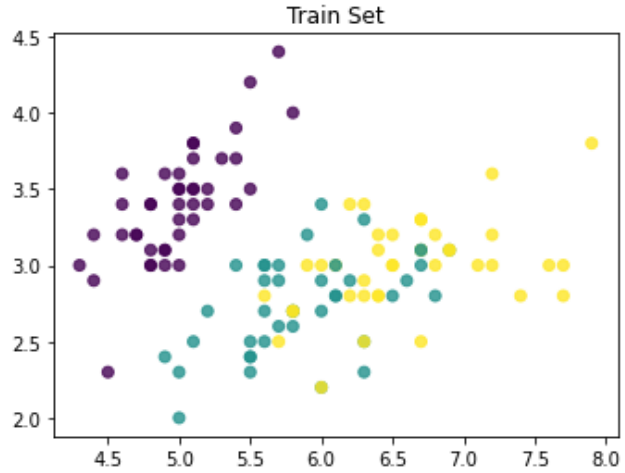
Ce graphique montre la répartition des données utilisées pour entraîner le modèle.

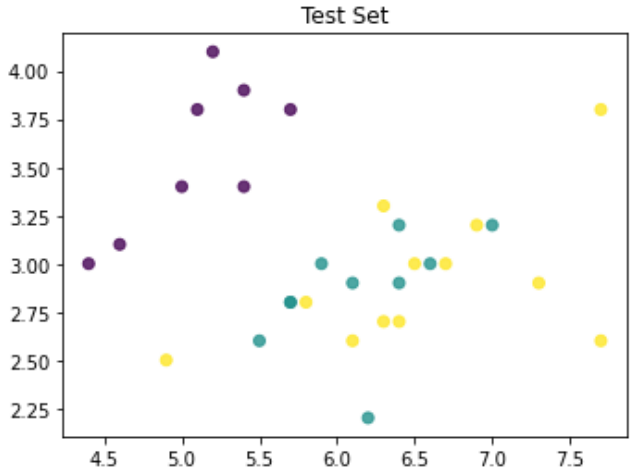
Les données sont colorées selon leurs classes et couvrent une large partie de l'espace des caractéristiques et cela garantit que le modèle dispose d'exemples variés pour apprendre

Les données de test sont distinctes de celles de l'entraînement et représentent une plus petite proportion.

La répartition des points permet de s'assurer que les classes sont également bien représentées dans l'évaluation du modèle.

b)





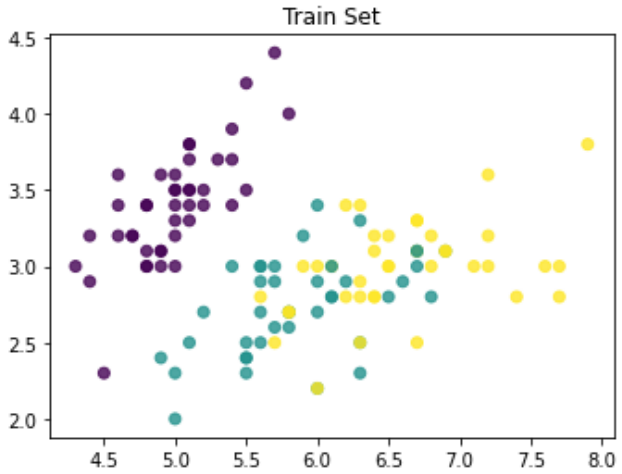
Après le relancement de « a » on trouve :

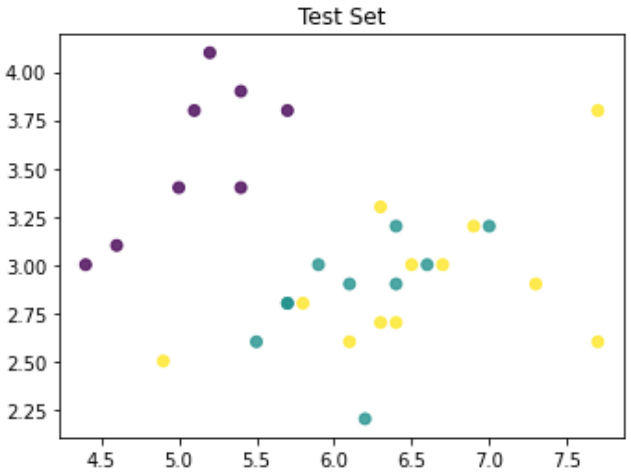
On constate que la répartition est changée par rapport à la première à cause de random_state qui garantit une distribution homogène des données

8. Pour la construction du jeu d'entrainement et de test, on ajoute le paramètre « random_state » :

```
X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.5, random_state=5)
plt.figure(figsize=(12, 4))
plt.subplot(121)
plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_train, alpha=0.8)
plt.title('Train Set')
plt.subplot(122)
plt.scatter(X_test[:, 0], X_test[:, 1], c=Y_test, alpha=0.8)
plt.title('Test Set')
```

résultat :





Apres le relancement et en utilisant le random-state, on peut remarquer que random-state sert à rendre le tirage aléatoire, sauf que la graine est toujours la même

Exercice 2 : Entrainement et évaluation du modèle

On s'intéresse à l'entrainement et à l'évaluation d'un modèle de machine learning en faisant appel à la bibliothèque sklearn. On choisit le modèle KNeighborsClassifier :

from sklearn.neighbors import KNeighborsClassifier

1. On définit le modèle comme suit :

   model = KNeighborsClassifier(1) en fixant le nombre de voisin à un.

2. Pour entrainer correctement le modèle, faites passer X_train et X_test dans la méthode fit comme suit : model.fit(X_train, y_train)

3. Evaluation du modèle, en utilisant « model.score » :

a).

```
# Création du modèle avec 1 voisin
model = KNeighborsClassifier(n_neighbors=1)
model.fit(X_train, y_train)
train_score = model.score(X_train, y_train)
print("Score sur les données d'entraînement :", train_score)
```

```
Score sur les données d'entraînement : 1.0
```

Le score obtenu mesure la proportion d'exemples correctement classés par le modèle sur les données d'entraînement. Un score de 1 donc 100% indique que le modèle a "mémorisé" les données.

b) sur les données qui n'ont pas été vues par le modèle.

```
test_score = model.score(X_test, y_test)
print("Score sur les données de test :", test_score)
```

```
Score sur les données de test : 0.9
```

Un score de **0.9 (90%)** sur les données de test signifie que le modèle a correctement classé 90 % des exemples qu'il n'avait jamais vus, cela montre une bonne capacité de généralisation et ce score montre que le modèle fonctionne bien.

Exercice 3 : Amélioration de l'entrainement et évaluation du modèle

1) Comparaison des performances du modèle lorsque le nombre de voisins est égal à 3 puis à 4 et Spécifier le jeu de données sur lequel on a comparé les performances.

Le code utilisé :

```python
# Importation des bibliothèques nécessaires
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score, validation_curve
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Chargement du jeu de données Iris
iris = load_iris()
X = iris.data
y = iris.target

# Étape 1 : Découpage des données
# 80% pour l'entraînement, 20% pour le test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=5)

# Découpage des 80% d'entraînement : 70% pour l'entraînement final, 30% pour la validation
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train, y_train, test_size=0.3, random_state=5)

# Étape 2 : Comparaison des performances pour k=3 et k=4
# Modèle avec k=3
model_k3 = KNeighborsClassifier(n_neighbors=3)
model_k3.fit(X_train_final, y_train_final)
val_score_k3 = accuracy_score(y_val, model_k3.predict(X_val))

# Modèle avec k=4
model_k4 = KNeighborsClassifier(n_neighbors=4)
model_k4.fit(X_train_final, y_train_final)
val_score_k4 = accuracy_score(y_val, model_k4.predict(X_val))

# Affichage des scores pour la validation
print(f"Score validation avec k=3 : {val_score_k3}")
print(f"Score validation avec k=4 : {val_score_k4}")
```

Remarque : Chaque partie dans le code est commentée.

Résultat :

```
Score validation avec k=3 : 0.9722222222222222
Score validation avec k=4 : 0.9166666666666666
```

Avec un score de **97,22 %** pour k=3 , le modèle est performant mais en augmentant le modèle devient moins performant comme illustré le résultat pour k=4 on obtient un score de 0,916.

Le jeu de test est toujours utilisé pour évaluer la performance finale, les données de test doivent rester inconnues pour le modèle pendant l'entraînement et la validation.

2) Modification du découpage du jeu de données :

```python
# Étape 1 : Découpage des données
# 60% pour l'entraînement, 40% pour le test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=5)

# Découpage des 80% d'entraînement : 70% pour l'entraînement final, 30% pour la validation
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train, y_train, test_size=0.3, random_state=5)
```

Résultat :

```
Score validation avec k=3 : 0.9629629629629629
Score validation avec k=4 : 0.9629629629629629
```

Un découpage différent a modifié les performances en fonction de la répartition des données, car on a obtenu un score diffèrent en prenant k=3 .

3) Pour comparer deux modèles, l'entrainement et la validation s'appuie sur la stratégie de la cross-validation. Elle consiste à entrainer puis valider le modèle sur plusieurs découpes possibles du jeu de données.

a)

```python
# Étape 4 : Validation croisée pour k=3 et k=4
cv_score_k3 = cross_val_score(KNeighborsClassifier(n_neighbors=3), X_train_final, y_train_final, cv=5, scoring='accuracy').mean()
cv_score_k4 = cross_val_score(KNeighborsClassifier(n_neighbors=4), X_train_final, y_train_final, cv=5, scoring='accuracy').mean()

print(f"Score moyen en validation croisée pour k=3 : {cv_score_k3}")
print(f"Score moyen en validation croisée pour k=4 : {cv_score_k4}")
```

La cross-validation permet de maximiser l'utilisation des données d'entraînement en divisant ces données en plusieurs sous-ensembles pour l'entraînement et la validation.

b) Le score obtenu :

```
Score moyen en validation croisée pour k=3 : 0.9522058823529411
Score moyen en validation croisée pour k=4 : 0.9522058823529411
```

c) Code pour la moyenne des 5 scores obtenus :

```python
# Étape 5 : Validation croisée pour plusieurs valeurs de k
val_scores = []
for k in range(1, 50):  # Tester k de 1 à 20
    score = cross_val_score(KNeighborsClassifier(n_neighbors=k), X_train_final, y_train_final, cv=5, scoring='accuracy').mean()
    val_scores.append(score)
```

d) Tracé des scores en fonction du nombre de voisins fixés :

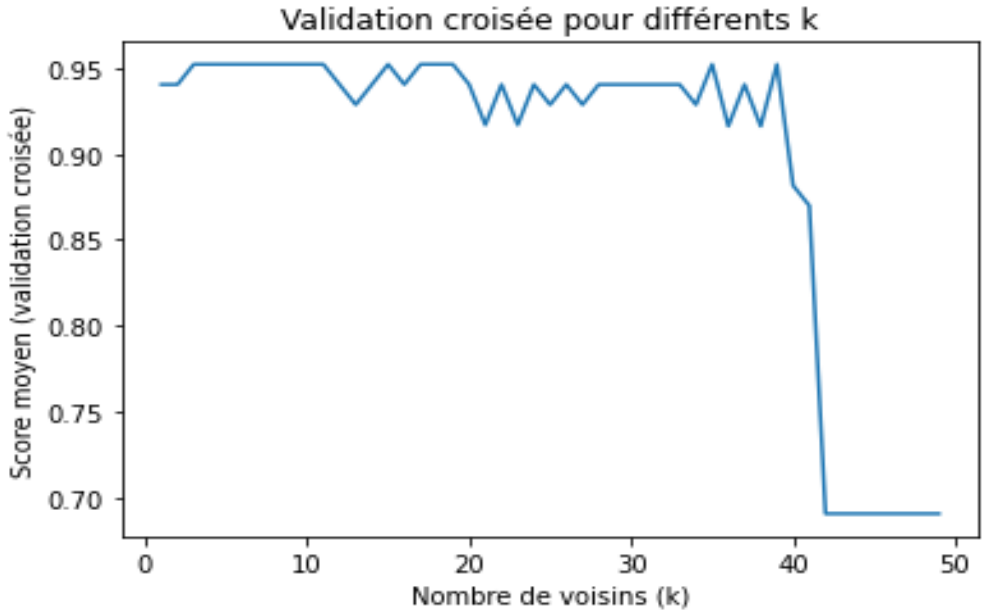
Code utilisé :

```
# Tracer les scores
plt.plot(range(1, 50), val_scores)
plt.xlabel('Nombre de voisins (k)')
plt.ylabel('Score moyen (validation croisée)')
plt.title('Validation croisée pour différents k')
plt.show()
```

Résultat obtenu :

```
Meilleur k : 3, avec un score moyen de 0.96
```

c'est le modèle au voisinage 4 car on a remarqué qu'il a le plus grand score.



e) Vérification des dimensions en utilisant :

```
# Meilleur k
best_k = range(1, 50)[val_scores.index(max(val_scores))]
print(f"Meilleur k : {best_k}, avec un score moyen de {max(val_scores):.2f}")
```

f) Scores moyens

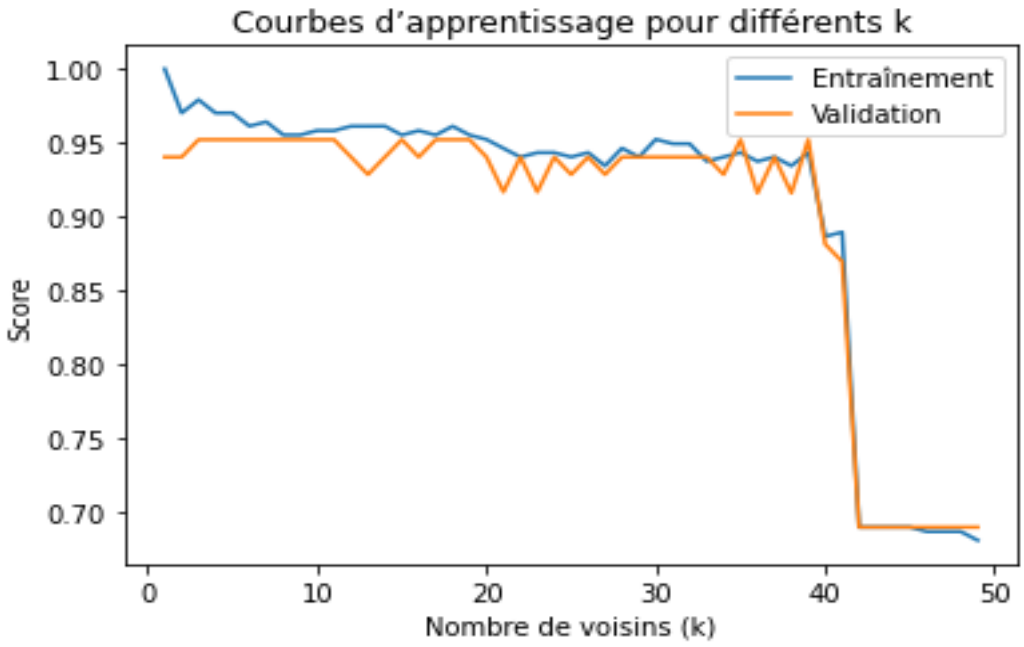
```
# Moyennes des scores
train_scores_mean = train_scores.mean(axis=1)
val_scores_mean = val_scores_curve.mean(axis=1)
```

g et h) Affichage des scores obtenus avec le jeu d'entrainement et avec le jeu de test :

```
# Étape 6 : Courbes d'apprentissage avec validation_curve
k_range = np.arange(1, 50)
train_scores, val_scores_curve = validation_curve(
    KNeighborsClassifier(),
    X_train_final,
    y_train_final,
    param_name='n_neighbors',
    param_range=k_range,
    cv=5
)
```

```
# Tracer les courbes
plt.plot(k_range, train_scores_mean, label='Entraînement')
plt.plot(k_range, val_scores_mean, label='Validation')
plt.xlabel('Nombre de voisins (k)')
plt.ylabel('Score')
plt.title('Courbes d'apprentissage pour différents k')
plt.legend()
plt.show()
```

Résultat :



On remarque qu'on a pas un grand écart entre l'entrainement et la validation de k=20 au 30

Au début l'écart un peu important à cause de overfitting .

Exercice 4 : Optimisation du modèle : recherche des meilleurs hyper paramètres :

1) Création d'un dictionnaire contenant les différents hyper paramètres :

```
#exo4
from sklearn.model_selection import GridSearchCV


param_grid ={'n_neighbors':np.arange(1,50),'metric': ['euclidian','manhattan']}
```

⇨ Le dictionnaire param_grid permet d'indiquer la meilleure valeur pour k et la métrique de distance (euclidean ou manhattan).

2) Grille avec plusieurs estimateurs :

```
Grid=GridSearchCV(KNeighborsClassifier(),param_grid,cv=5)
```

3) Entrainenement du modele avec (Grid.fit(X_train ,Y_train)

```
Grid.fit(X_train,Y_train)
```

4) Les parametres du modele :

```
Meilleurs paramètres : {'metric': 'manhattan', 'n_neighbors': 1}
Meilleur score de validation : 0.975
```

5) Model_grid.best_estimator

```
# Sauvegarde du meilleur modèle
best_model = Grid.best_estimator_

print("Meilleur modèle :", best_model)
```

```
Meilleur modèle : KNeighborsClassifier(metric='manhattan', n_neighbors=1)
```

6) Evaluation des performances du modèle

```
# Évaluation des performances sur le jeu de test
test_score = best_model.score(X_test, y_test)
print("Score sur le jeu de test :", test_score)
```

```
Score sur le jeu de test : 0.8666666666666667
```

Interprétations :

⇨ Best_model.score représente le score moyen en validation croisée pour les meilleurs hyperparamètres.

⇨ Si ce score est similaire à meilleur score de validation cela signifie que y'a pas de sur ou sous apprentissage

7) Mesurer les performances au moyen de la matrice de confusion

```
# Calcul de la matrice de confusion
conf_matrix = confusion_matrix(y_test, best_model.predict(X_test))

print("Matrice de confusion :\n", conf_matrix)
```

```
Matrice de confusion :
 [[8 0 0]
 [0 9 2]
 [0 2 9]]
```

Interprétations :

- Les **valeurs sur la diagonale** correspondent au nombre de prédictions correctes pour chaque classe
- Les **valeurs hors diagonale** indiquent les erreurs de classification par exemple combien le modèle a pris de C0 de C1(prédis) et qui normalement va être 0.
- On remarque que les valeurs de diagonale sont fortes.
- On remarque que notre modèle a 30 exemples.

Exercice 5 : Interprétation des courbes d'apprentissage

1) **N** représente le nombre de fractions (ou de sous-ensembles) dans lequel le jeu de données d'entraînement est découpé. Dans ce cas, vous avez spécifié que **N = 10**, ce qui signifie que les courbes seront tracées en utilisant 10 tailles d'échantillons différentes allant de 10 % à 100 % des données d'entraînement

```
# Génération des courbes d'apprentissage
N, train_scores, val_scores = learning_curve(
    best_model,
    X_train,  # Données d'entraînement
    y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),  # 10 lots de tailles croissa
    cv=5  # Cross-validation à 5 folds
)
```

- **learning_curve** : Permet de générer les scores pour différentes tailles de jeu d'entraînement
- train_sizes divise les données en 10 lots, de 10% à 100%
- cv=5 applique une cross-validation pour évaluer les performances sur chaque lot

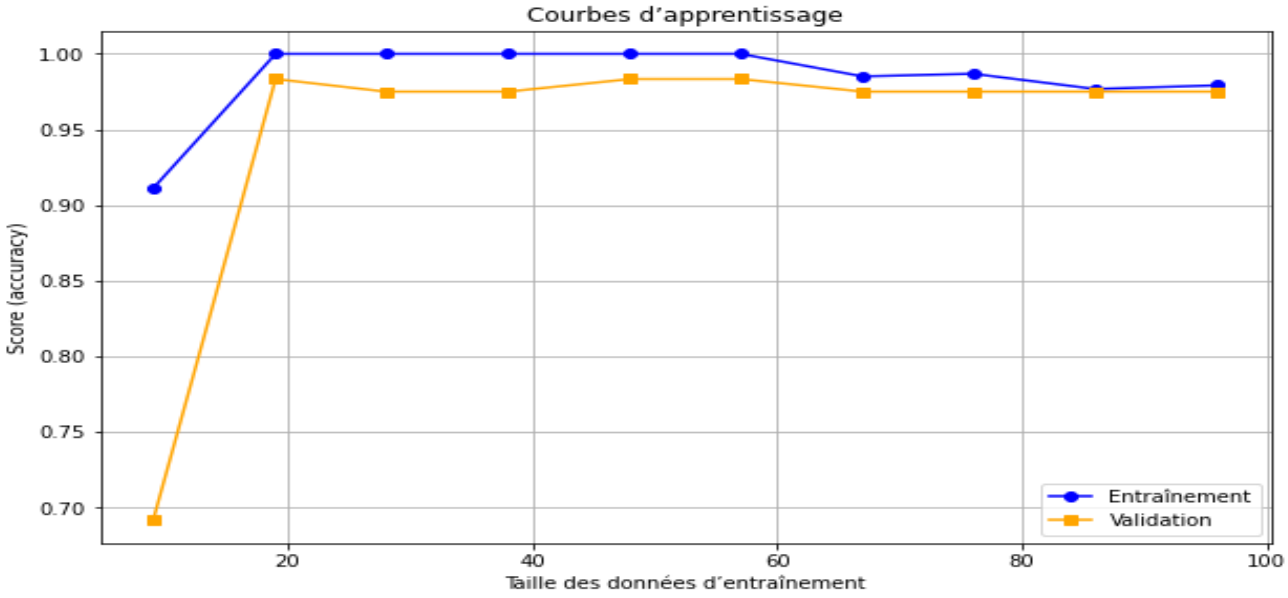
2) + 3) Affichez les scores moyens sur le jeu de données de validation et entrainement en utilisant :

```
)

# Moyennes des scores
train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)

# Visualisation des courbes
plt.figure(figsize=(10, 6))

# Courbe pour le jeu d'entraînement
plt.plot(N, train_mean, label="Entraînement", color="blue", marker="o")

# Courbe pour le jeu de validation
plt.plot(N, val_mean, label="Validation", color="orange", marker="o")

# Légendes et titres
plt.title("Courbes d'apprentissage")
plt.xlabel("Taille des données d'entraînement")
plt.ylabel("Score (Accuracy)")
plt.legend()
plt.grid()
plt.show()
```

- On calcule la moyenne des scores pour l'entraînement (train_mean) et la validation (val_mean).
- On trace deux courbes :
  La performance sur les données d'entraînement.
  La performance sur les données de validation.

3) Interprétez les courbes obtenues. Est-il nécessaire de collecter plus de données ?

Les courbes d’apprentissages ci-dessous :



4) Interprétations

- ❖ L’écart entre les courbes est faible, le modèle est bien généralisé. Un écart important pourrait indiquer un problème (sous-apprentissage ou sur-apprentissage).
- ❖ Les courbes atteignent un plateau, cela indique qu'ajouter plus de données d'entraînement n'apportera pas d'amélioration significative comme illustré l’extrémité des deux courbes sont presque superposées.