Tp_2_Arbre_Thomas_Laugié

September 27, 2023

***TP_2_Arbre* Thomas Laugié**

Question 1

Lorsque vous effectuez une régression à l'aide d'arbres de décision, une mesure d'homogénéité couramment utilisée est le MSE (Mean Squared Error). C'est la méthode par défaut dans DecisionTreeRegressor de Scikit-Learn. Une autre mesure est le MAE (Mean Absolute Error). La différence principale entre MSE et MAE réside dans la manière dont ils pénalisent les erreurs : MSE pénalise davantage les grandes erreurs que le MAE.

Question 2

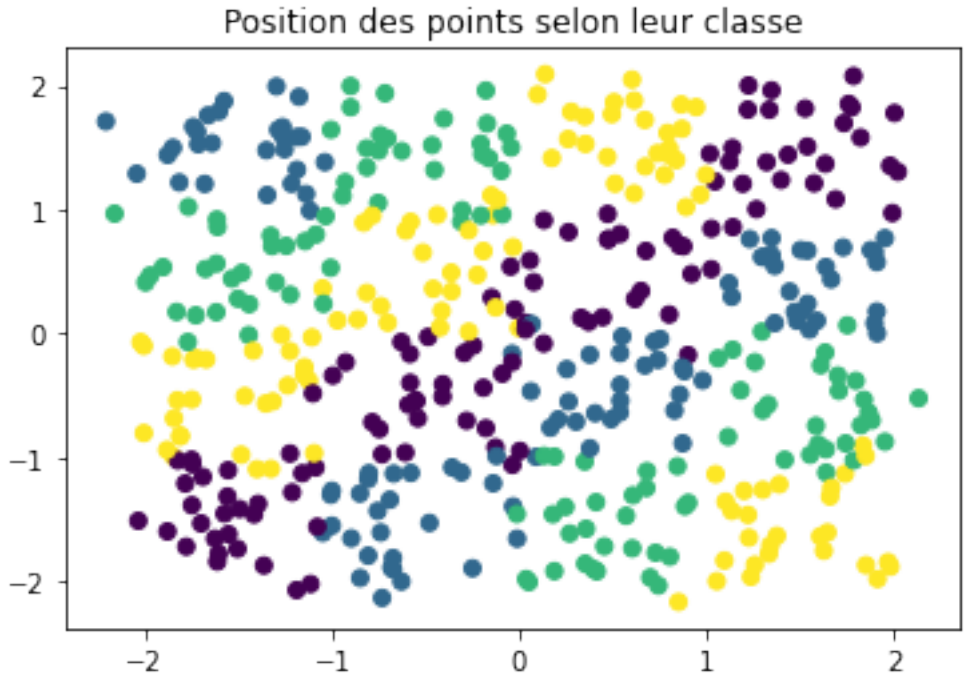
Simulation d'échantillons de données :

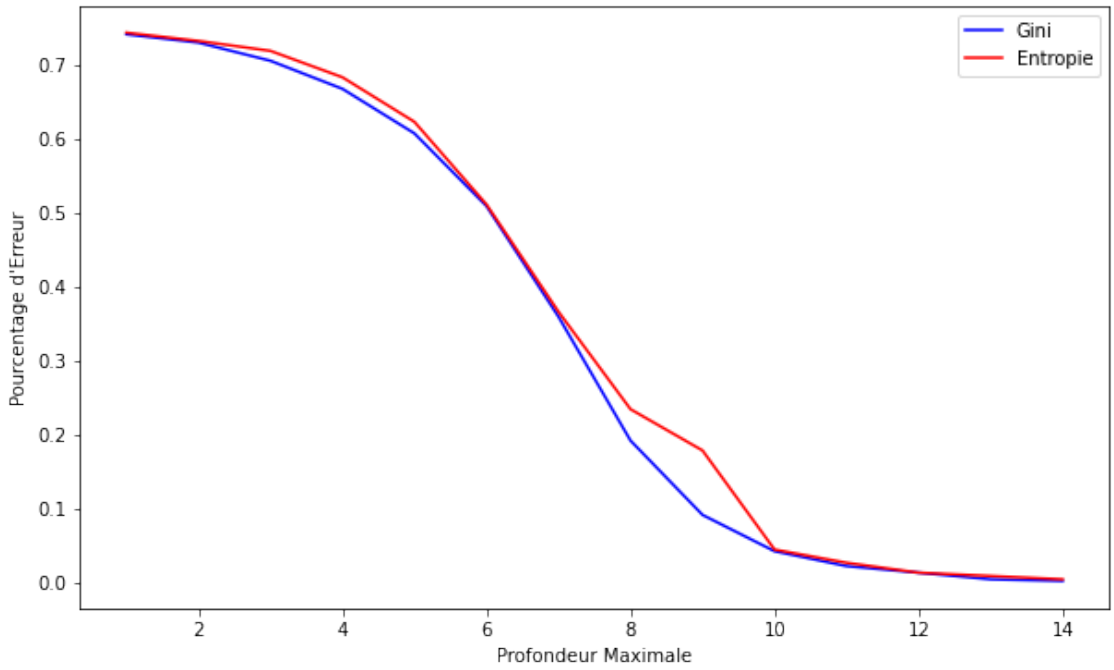
On utilisation de la fonction rand_checkers pour simuler un échantillon de taille n=456, ce qui permet un équilibrage des classes pour assurer une distribution uniforme.

Pour chaque profondeur d'arbre, allant de 1 à 14, deux arbres de décision sont entraînés : l'un avec le critère Gini et l'autre avec l'Entropie. Pour chaque arbre, après entraînement, le taux d'erreur est calculé sur l'ensemble de données.

[1]:
```
from sklearn import tree
from sklearn import metrics
from tp_arbres_source import rand_checkers
import matplotlib.pyplot as plt
import numpy as np


data = rand_checkers(n1=114, n2=114, n3=114, n4=114)
X, y = data[:, :2], data[:, 2]

plt.scatter(data[:,0], data[:,1], c=data[:,2])
plt.title('Position des points selon leur classe')
plt.show()

depths = range(1, 15)
errors_gini = []
errors_entropy = []

for depth in depths:
```

```
    for criterion in ["gini", "entropy"]:
        clf = tree.DecisionTreeClassifier(criterion=criterion, max_depth=depth)
        clf.fit(X, y)
        y_pred = clf.predict(X)
        error = 1 - metrics.accuracy_score(y, y_pred)

        if criterion == "gini":
            errors_gini.append(error)
        else:
            errors_entropy.append(error)

plt.figure(figsize=(10, 6))
plt.plot(depths, errors_gini, label='Gini', color='blue')
plt.plot(depths, errors_entropy, label='Entropie', color='red')
plt.xlabel('Profondeur Maximale')
plt.ylabel('Pourcentage d\'Erreur')
plt.legend()
plt.show()
```





On a donc produit deux courbes permettant de comparer l'évolution du pourcentage d'erreurs en fonction de la profondeur de l'arbre, selon les critères Gini et Entropie.

Ces courbes offrent une vue d'ensemble sur les performances des arbres en fonction de leur complexité et du critère de division choisi.

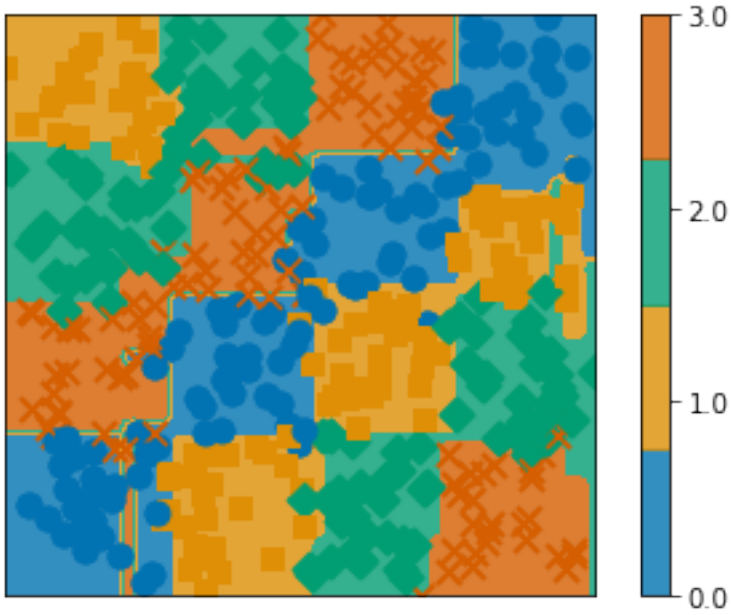
On observe une forte baisse au début et qui s'atténue fortement vers 10 de profondeur.

Question 3

On identifie la profondeur qui donne le pourcentage d'erreurs le plus bas pour le critère d'entropie. Pour ce faire, on recherche la valeur minimale dans la liste errors_entropy et on identifie la profondeur correspondante. On va entraîner un arbre de décision avec cette profondeur optimale en utilisant le critère d'entropie et utiliser les fonctions plot_2d pour afficher les données et frontiere pour montrer la séparation des classes réalisée par l'arbre entraîné.

[2]:
```
from tp_arbres_source import frontiere,plot_2d

best_depth_entropy = depths[np.argmin(errors_entropy)]
clf_best = tree.DecisionTreeClassifier(criterion="entropy",␣
 ↪max_depth=best_depth_entropy)
clf_best.fit(X, y)

def predict_aux(input_1d):
    return clf_best.predict(input_1d.reshape(1, -1))

plot_2d(X, y)
```

```
frontiere(predict_aux, X, y)
plt.show()
```

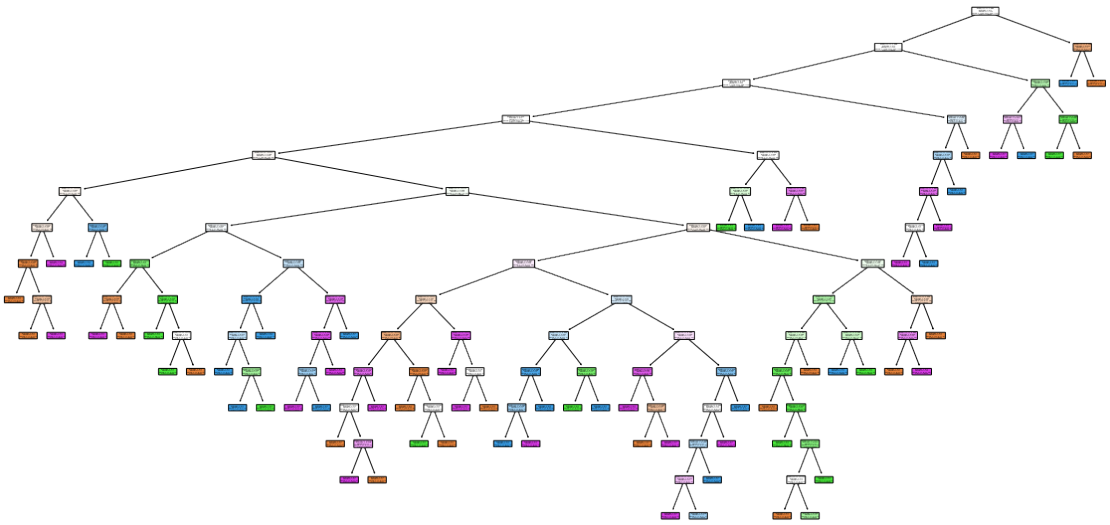


On remarque une bonne classifiquation et de bonne frontière avec la profondeur maximal.

Question 4

On visualiser le graphique de l'arbre de décision obtenu à la question précédente.

[3]:
```
plt.figure(figsize=(20, 10))
tree.plot_tree(clf_best, filled=True, feature_names=["feature1", "feature2"],␣
 ↪class_names=["class1", "class2","class3","class4"], rounded=True)
plt.savefig('arbre_decision.pdf')
plt.show()
```



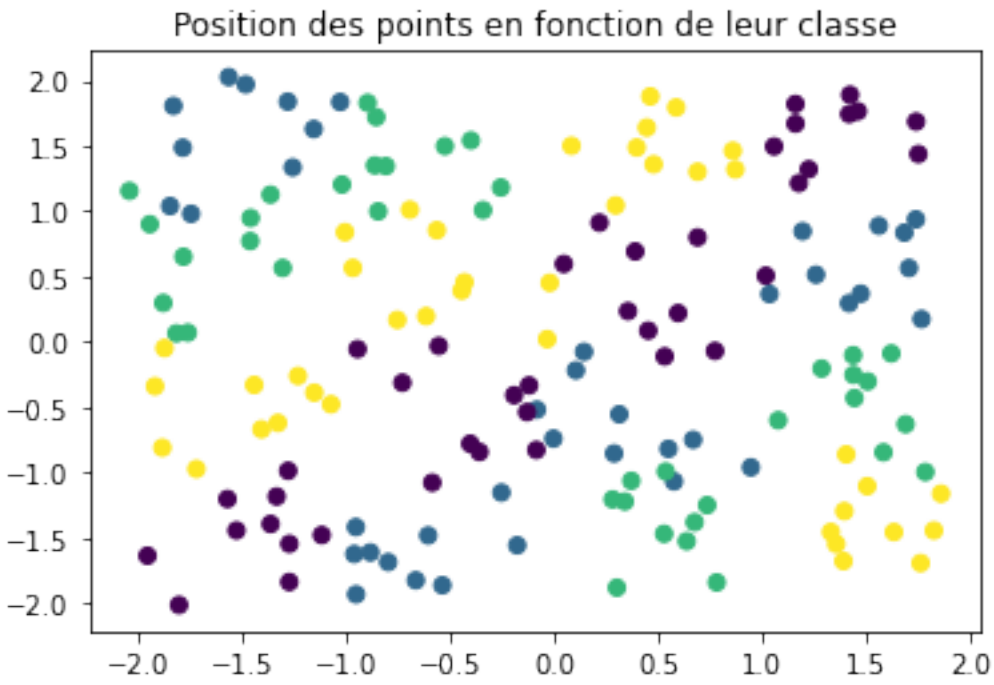
Le code génère une représentation visuelle détaillée de l'arbre de décision, avec les noms des caractéristiques, les classes et les décisions à chaque nœud. Chaque nœud indique la caractéristique utilisée pour la division, la pureté (gini ou entropie), le nombre d'échantillons, et la distribution des classes.

Mais avec la profondeur maximal l'arbre est trop grand et on ne peut pas bien distingué.

Question 5

On génére un nouvel échantillon de test de taille n=160, où chaque classe est représentée par 40 échantillons. Puis on prédit les classes de cet échantillon de test en utilisant l'arbre de décision précédemment entraîné.

[4]:
```
n = 160
data_new = rand_checkers(n1=n//4, n2=n//4, n3=n//4, n4=n//4)  # Cela donnera un␣
↪total de 456 échantillons
X_new, y_new = data_new[:, :2], data_new[:, 2]


plt.scatter(data_new[:,0],data_new[:,1],c=data_new[:,2])
plt.title('Position des points en fonction de leur classe')
plt.show()

y_pred = clf_best.predict(X_new)

error_rate = sum(y_pred != y_new) / len(y_new)
print(f"Proportion d'erreurs sur le nouvel ensemble de test : {error_rate:.2f}")
```




```
Proportion d'erreurs sur le nouvel ensemble de test : 0.19
```

La proportion d'erreurs sur cet échantillon de test vous donne une indication sur la capacité du modèle à généraliser à de nouvelles données. Si le taux d'erreur est élevé, cela peut signifier que votre modèle est sur-ajusté (overfitting) sur les données d'entraînement. Si le taux d'erreur est raisonnablement bas, cela signifie que votre modèle est capable de bien généraliser à de nouvelles données.

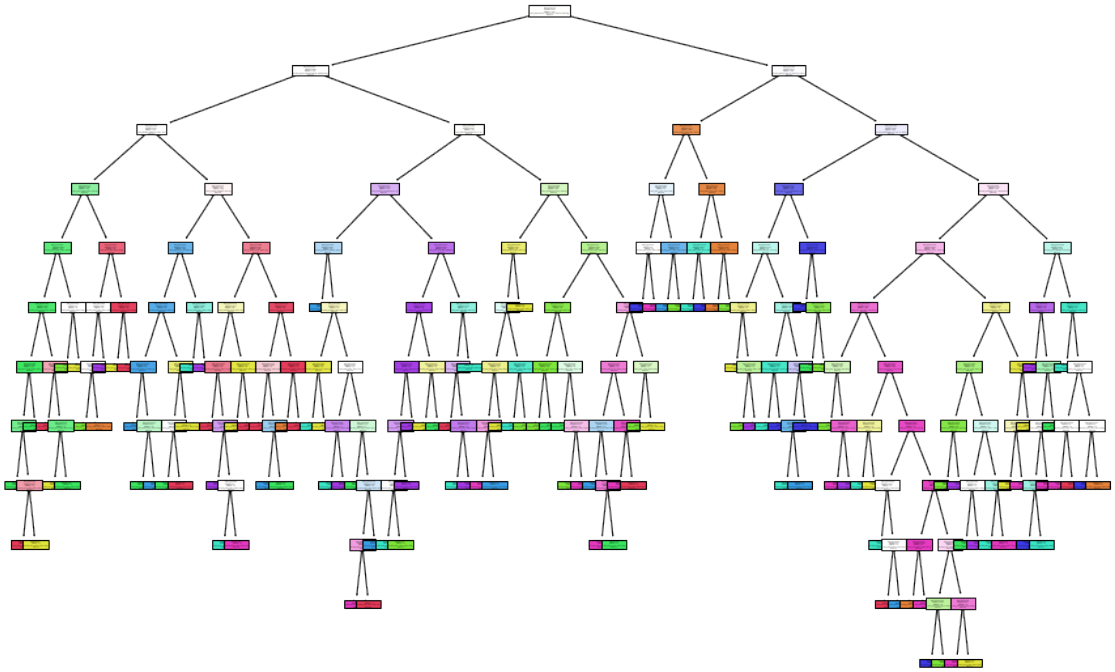
Question 6

Le jeu de données "digits" est importé depuis le module sklearn.datasets. Ce dataset contient des images de chiffres écrits à la main, où chaque image est représentée par un vecteur de 64 pixels, et la cible est le chiffre correspondant (0 à 9).

On crée un arbre de décision utilisant l'entropie comme critère est entraîné pour différents niveaux de profondeur. L'objectif est d'identifier la profondeur qui minimise l'erreur sur l'ensemble de test et de calculer la proportion d'erreurs obtenue sur l'ensemble de test.

[5]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt
import numpy as np

digits = load_digits()
X, y = digits.data, digits.target
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)


depths = list(range(1, 21))
errors_entropy = []

for depth in depths:
    clf = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
    clf.fit(X_train, y_train)
    errors_entropy.append(1 - clf.score(X_test, y_test))

best_depth_entropy = depths[np.argmin(errors_entropy)]


clf_best = DecisionTreeClassifier(criterion="entropy",␣
 ↪max_depth=best_depth_entropy)
clf_best.fit(X_train, y_train)


plt.figure(figsize=(15, 10))
plot_tree(clf_best, filled=True, feature_names=digits.feature_names,␣
 ↪class_names=[str(i) for i in digits.target_names])
plt.show()


y_pred = clf_best.predict(X_test)
error_rate = sum(y_pred != y_test) / len(y_test)
print(f"Proportion d'erreurs sur l'ensemble de test : {error_rate:.2f}")
```



```
Proportion d'erreurs sur l'ensemble de test : 0.14
```

La proportion d'erreurs obtenue sur l'ensemble de test est de 14%. Cela signifie que l'arbre de décision prédit correctement 88% des images dans l'ensemble de test, ce qui est une performance assez bonne pour un classificateur simple comme un arbre de décision.

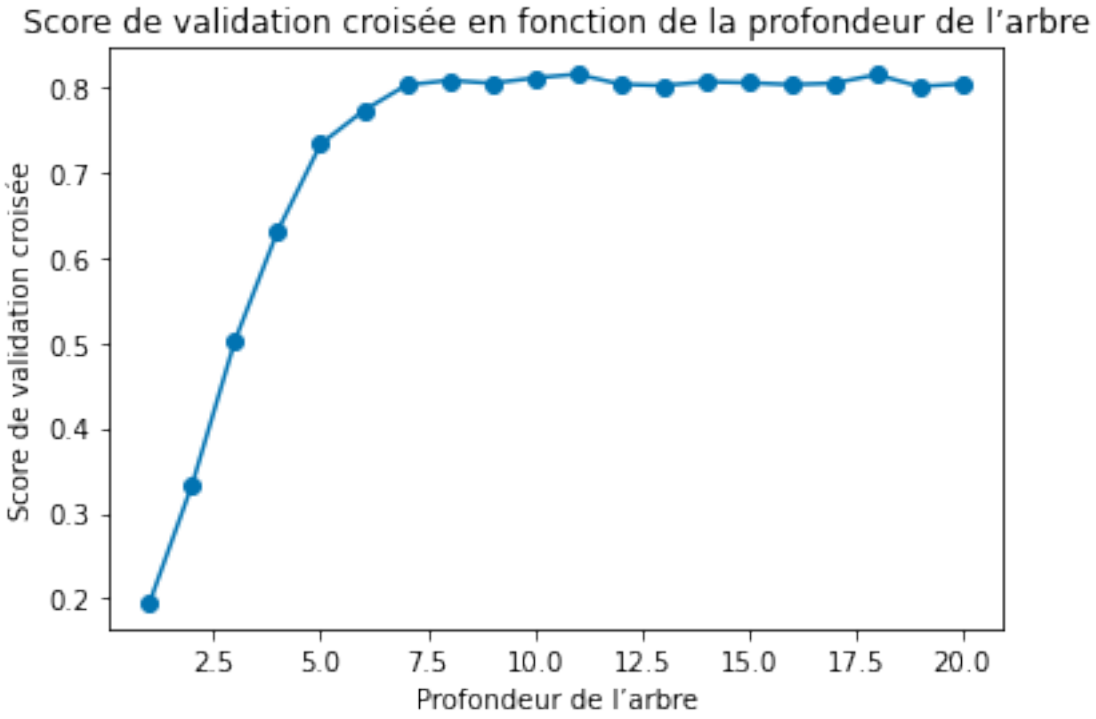
Question 7

Afin de choisir la meilleure profondeur pour l'arbre de décision pour le jeu de données "digits", nous utilisons la validation croisée. C'est une méthode qui permet d'estimer la performance d'un modèle en divisant l'ensemble de données en plusieurs sous-ensembles (folds) et en entraînant/testant le modèle sur ces sous-ensembles de manière répétée.

[6]:
```
from sklearn.model_selection import cross_val_score


depths = list(range(1, 21))
mean_scores = []

for depth in depths:
    clf = DecisionTreeClassifier(criterion="entropy", max_depth=depth)
    scores = cross_val_score(clf, X, y)  # Utilisation d'une validation croisée␣
 ↪5-fold
    mean_scores.append(np.mean(scores))
```

```
best_depth = depths[np.argmax(mean_scores)]

print(f"La profondeur optimale est : {best_depth}")
print(f"Le score moyen correspondant est : {max(mean_scores):.4f}")


import matplotlib.pyplot as plt
plt.plot(depths, mean_scores, marker='o')
plt.xlabel('Profondeur de l'arbre')
plt.ylabel('Score de validation croisée')
plt.title('Score de validation croisée en fonction de la profondeur de l'arbre')
plt.show()
```

```
La profondeur optimale est : 11
Le score moyen correspondant est : 0.8147
```



La profondeur optimale trouvée est de 11. Avec cette profondeur, le score moyen de validation croisée, obtenu sur l'ensemble du dataset, est de 81.47%. Le graphique illustre l'évolution du score moyen de validation croisée en fonction de la profondeur de l'arbre.

En conclusion, pour le jeu de données "digits", un arbre de décision avec une profondeur de 15 semble être le choix optimal pour obtenir de bonnes performances.

Question 8

L'objectif de cette question est d'afficher la courbe d'apprentissage (learning curve) pour les arbres de décisions sur un jeu de données donné, en s'inspirant d'un exemple fourni par scikit-learn.

[14]:
```
from sklearn.model_selection import learning_curve
from sklearn.datasets import load_digits
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

naive_bayes = GaussianNB()
svc = SVC(kernel="rbf", gamma=0.001)

common_params = {
    "X": X,
    "y": y,
    "train_sizes": np.linspace(0.1, 1.0, 5),
    "cv": ShuffleSplit(n_splits=50, test_size=0.2, random_state=0),
    "n_jobs": 4,
    "return_times": True,
}

train_sizes, _, test_scores_nb, fit_times_nb, score_times_nb = learning_curve(
    naive_bayes, **common_params
)
train_sizes, _, test_scores_svm, fit_times_svm, score_times_svm = learning_curve(
    svc, **common_params
)

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(16, 12), sharex=True)

for ax_idx, (fit_times, score_times, estimator) in enumerate(
    zip(
        [fit_times_nb, fit_times_svm],
        [score_times_nb, score_times_svm],
        [naive_bayes, svc],
    )
):
    # scalability regarding the fit time
    ax[0, ax_idx].plot(train_sizes, fit_times.mean(axis=1), "o-")
    ax[0, ax_idx].fill_between(
        train_sizes,
        fit_times.mean(axis=1) - fit_times.std(axis=1),
        fit_times.mean(axis=1) + fit_times.std(axis=1),
        alpha=0.3,
    )
    ax[0, ax_idx].set_ylabel("Fit time (s)")
    ax[0, ax_idx].set_title(
        f"Scalability of the {estimator.__class__.__name__} classifier"
```

```
    )

    # scalability regarding the score time
    ax[1, ax_idx].plot(train_sizes, score_times.mean(axis=1), "o-")
    ax[1, ax_idx].fill_between(
        train_sizes,
        score_times.mean(axis=1) - score_times.std(axis=1),
        score_times.mean(axis=1) + score_times.std(axis=1),
        alpha=0.3,
    )
    ax[1, ax_idx].set_ylabel("Score time (s)")
    ax[1, ax_idx].set_xlabel("Number of training samples")
```