Classification d'iris avec un arbre de décision

Dans ce notebook, nous allons utiliser le **jeu de données Iris**, très connu en machine learning, pour entraîner un **arbre de décision** à reconnaître l'espèce d'une fleur à partir de ses caractéristiques.

Nous allons passer par toutes les étapes classiques :

1. Chargement des données
2. Préparation des données
3. Séparation entre entraînement et test
4. Entraînement du modèle
5. Évaluation
6. Visualisation de l'arbre
7. Prédiction sur un exemple

---

1. Importation des bibliothèques nécessaires

In [1]:
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn import tree
import matplotlib.pyplot as plt
```

2. Chargement du jeu de données Iris

Le dataset contient 150 exemples de fleurs avec 4 caractéristiques :

- longueur du sépale
- largeur du sépale
- longueur du pétale
- largeur du pétale

Chaque fleur appartient à une des trois espèces :

- **setosa**
- **versicolor**
- **virginica**

In [2]:
```
iris = datasets.load_iris()
X = iris.data  # Les caractéristiques (features)
y = iris.target  # Les étiquettes (classes)
```

3. Séparation des données en jeu d'entraînement et de test

Nous allons utiliser 70% des données pour entraîner le modèle, et 30% pour le tester.

In [3]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_
```

4. Entraînement d'un arbre de décision

On entraîne un **DecisionTreeClassifier** de `scikit-learn` sur nos données d'entraînement.

In [4]:
```
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)
```

Out[4]:
```
DecisionTreeClassifier(random_state=42)
```

5. Prédiction et évaluation du modèle

On prédit les classes des données de test et on calcule la **précision** du modèle.

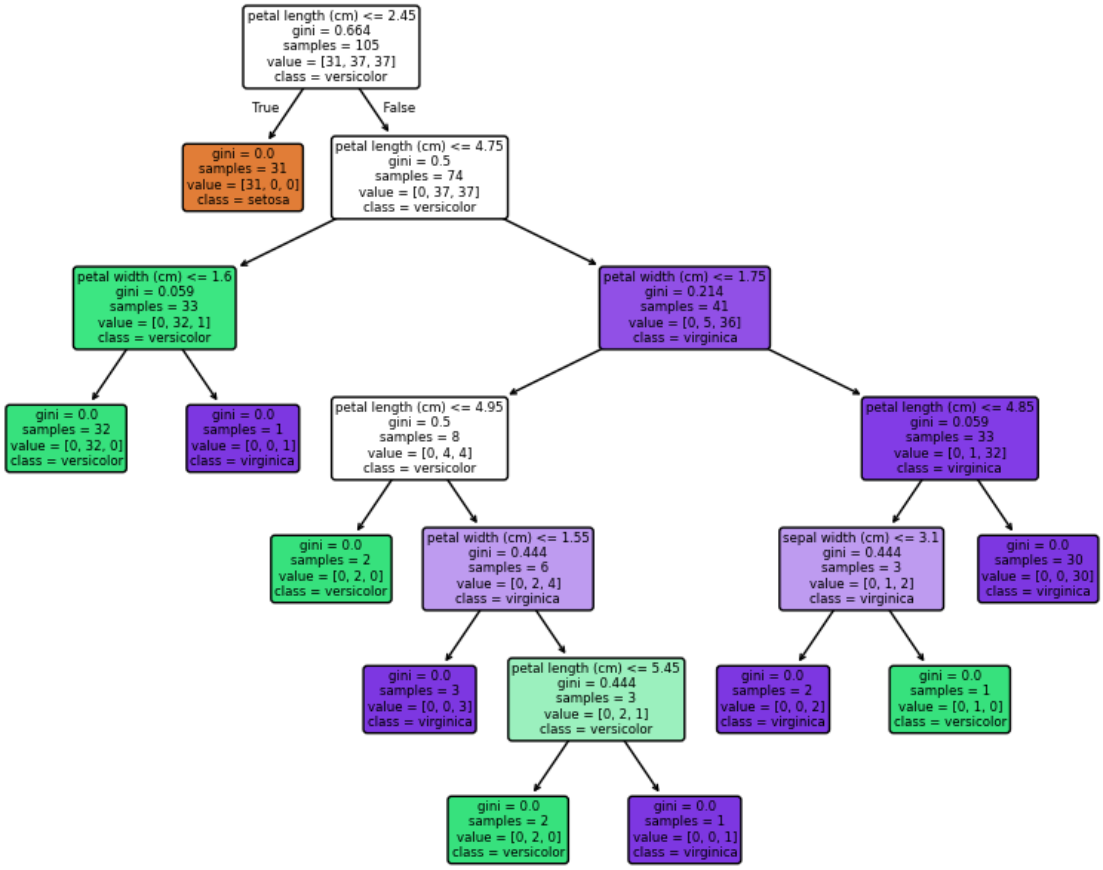
In [5]:
```
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Précision du classifieur : {accuracy * 100:.2f}%")
```

```
Précision du classifieur : 100.00%
```

6. Visualisation de l'arbre de décision

Une visualisation permet de mieux comprendre comment l'arbre prend ses décisions.

In [6]:
```
plt.figure(figsize=(10, 8))
tree.plot_tree(clf,
               filled=True,
               feature_names=iris.feature_names,
               class_names=iris.target_names,
               rounded=True)
plt.show()
```



7. Exemple de classification

On crée un nouvel exemple (une fleur avec certaines mesures) et on demande au modèle de prédire son espèce.

In [7]:
```
# Exemple : fleur avec ces caractéristiques
new_sample = [[5.5, 2.4, 3.8, 1.1]]

# Prédiction
predicted_class = clf.predict(new_sample)

# Affichage du résultat
print(f"Classe prédite pour l'échantillon : {iris.target_names[predicted_class][
```

```
Classe prédite pour l'échantillon : versicolor
```

1. Variation du paramètre `max_depth` et du critère (`criterion`)

Nous allons tester plusieurs valeurs pour le paramètre `max_depth` (profondeur maximale de l'arbre) et deux critères de séparation :

- **gini** (impureté de Gini)
- **entropy** (information gain)

Pour chaque combinaison, nous entraînerons le modèle et évaluerons sa précision sur le jeu de test.

In [9]:
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn import tree
import matplotlib.pyplot as plt
import pandas as pd

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

# Séparation en jeu d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_
```

In [10]:
```
# Définition des valeurs à tester
criteria = ['gini', 'entropy']
max_depth_values = [None, 2, 3, 4, 5, 6, 7]

results = []

for crit in criteria:
    for md in max_depth_values:
        clf_param = DecisionTreeClassifier(criterion=crit, max_depth=md, random_
        clf_param.fit(X_train, y_train)
        y_pred_param = clf_param.predict(X_test)
        acc = accuracy_score(y_test, y_pred_param)
        results.append({'Critère': crit, 'Max Depth': md, 'Précision': acc * 100

# Conversion des résultats en DataFrame pour affichage
df_results = pd.DataFrame(results)
print("Comparaison des paramètres max_depth et criterion :")
print(df_results)
```

```
Comparaison des paramètres max_depth et criterion :
    Critère  Max Depth   Précision
0      gini        NaN  100.000000
1      gini        2.0   97.777778
2      gini        3.0  100.000000
3      gini        4.0  100.000000
4      gini        5.0  100.000000
5      gini        6.0  100.000000
6      gini        7.0  100.000000
7   entropy        NaN   97.777778
8   entropy        2.0   97.777778
9   entropy        3.0   97.777778
10  entropy        4.0  100.000000
11  entropy        5.0   97.777778
12  entropy        6.0   97.777778
13  entropy        7.0   97.777778
```

Analyse des résultats pour `max_depth` et `criterion`

En observant le tableau ci-dessus, vous pouvez :

- Identifier si une profondeur limitée améliore la généralisation en évitant le sur-apprentissage.
- Comparer les performances entre le critère **gini** et **entropy**.

Par exemple, une valeur de `max_depth` trop faible peut entraîner une sous-apprentissage (précision trop faible), tandis qu'une valeur trop élevée peut rendre l'arbre trop complexe et favoriser le sur-apprentissage.
Il est intéressant de noter que pour ce jeu de données, la différence entre les critères n'est souvent pas très marquée.

2. Variation du paramètre `min_samples_split`

Ensuite, nous allons tester l'impact du paramètre `min_samples_split`, qui définit le nombre minimal d'échantillons requis pour diviser un nœud.
Un paramètre plus élevé peut rendre l'arbre plus robuste, mais trop élevé peut empêcher la capture de structures fines dans les données.

In [11]:
```
# Définition des valeurs pour min_samples_split à tester
min_samples_values = [2, 3, 4, 5, 6, 7, 8, 10]
results_split = []

for mss in min_samples_values:
    clf_param = DecisionTreeClassifier(min_samples_split=mss, random_state=42)
    clf_param.fit(X_train, y_train)
    y_pred_param = clf_param.predict(X_test)
    acc = accuracy_score(y_test, y_pred_param)
    results_split.append({'Min Samples Split': mss, 'Précision': acc * 100})

df_results_split = pd.DataFrame(results_split)
print("Comparaison des valeurs de min_samples_split :")
print(df_results_split)
```

```
Comparaison des valeurs de min_samples_split :
   Min Samples Split  Précision
0                  2      100.0
1                  3      100.0
2                  4      100.0
3                  5      100.0
4                  6      100.0
5                  7      100.0
6                  8      100.0
7                 10      100.0
```

Analyse des résultats pour `min_samples_split`

Le tableau ci-dessus montre comment la précision varie en fonction de la valeur de `min_samples_split` :

- Une valeur trop faible peut entraîner un arbre très complexe, susceptible de sur-apprentissage.

- Une valeur trop élevée peut simplifier excessivement l'arbre, menant à une sous-apprentissage.

Pour le jeu de données Iris, vous pouvez observer la valeur qui maximise la précision tout en maintenant une bonne capacité de généralisation.

3. Conclusion

En testant différentes combinaisons de paramètres, nous pouvons :

- **Optimiser le modèle** en choisissant des valeurs qui offrent un bon compromis entre complexité et performance.
- **Comprendre l'influence** de chaque paramètre sur la capacité du modèle à généraliser sur de nouvelles données.

Ces analyses nous permettent de mieux interpréter et ajuster les modèles d'apprentissage automatique pour obtenir des résultats robustes et fiables.

In [ ]: