

## ▼ The Iris Dataset: Decision Tree

### ▼ 1. Importing necessary libraries

First, we need to import the libraries required for our analysis: `pandas` for data manipulation, `sklearn` for building the decision tree and evaluating its performance, and `matplotlib` for visualising the decision tree and the results.

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
import matplotlib.pyplot as plt
```


### ▼ 2. Loading the dataset



We load the Iris dataset, which is a well-known dataset in machine learning that includes measurements of iris flowers. The features (input variables) are stored in `X`, and the target labels (species of the iris flowers) are stored in `y`. We convert this data into a Pandas `DataFrame`, which allows us to easily visualise and manipulate the data.

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

df = pd.DataFrame(data=X, columns=iris.feature_names)
df['target'] = y

# Displaying the first few rows of the DataFrame
df.head()
```



	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	
0	5.1	3.5	1.4	0.2	0	
1	4.9	3.0	1.4	0.2	0	
2	4.7	3.2	1.3	0.2	0	
3	4.6	3.1	1.5	0.2	0	
4	5.0	3.6	1.4	0.2	0	

Next steps:

[Generate code with df](#)

[View recommended plots](#)

[New interactive sheet](#)

### ▼ 3. Splitting the data into training and testing sets


We now split the dataset into training and testing sets using the `train_test_split` function. We specify `test_size=0.2` to reserve 20% of the data for testing, and `random_state=42` to ensure the data is split consistently every time we run the code.

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

### ▼ 4. Creating and training the decision tree

In this section, we create an instance of the decision tree classifier and train it. The model learns the relationship between the features (`X_train`) and the target labels (`y_train`).

```
clf_iris = DecisionTreeClassifier(random_state=42)
clf_iris.fit(X_train, y_train)
```



▼ DecisionTreeClassifier ⓘ ?

DecisionTreeClassifier(random\_state=42)

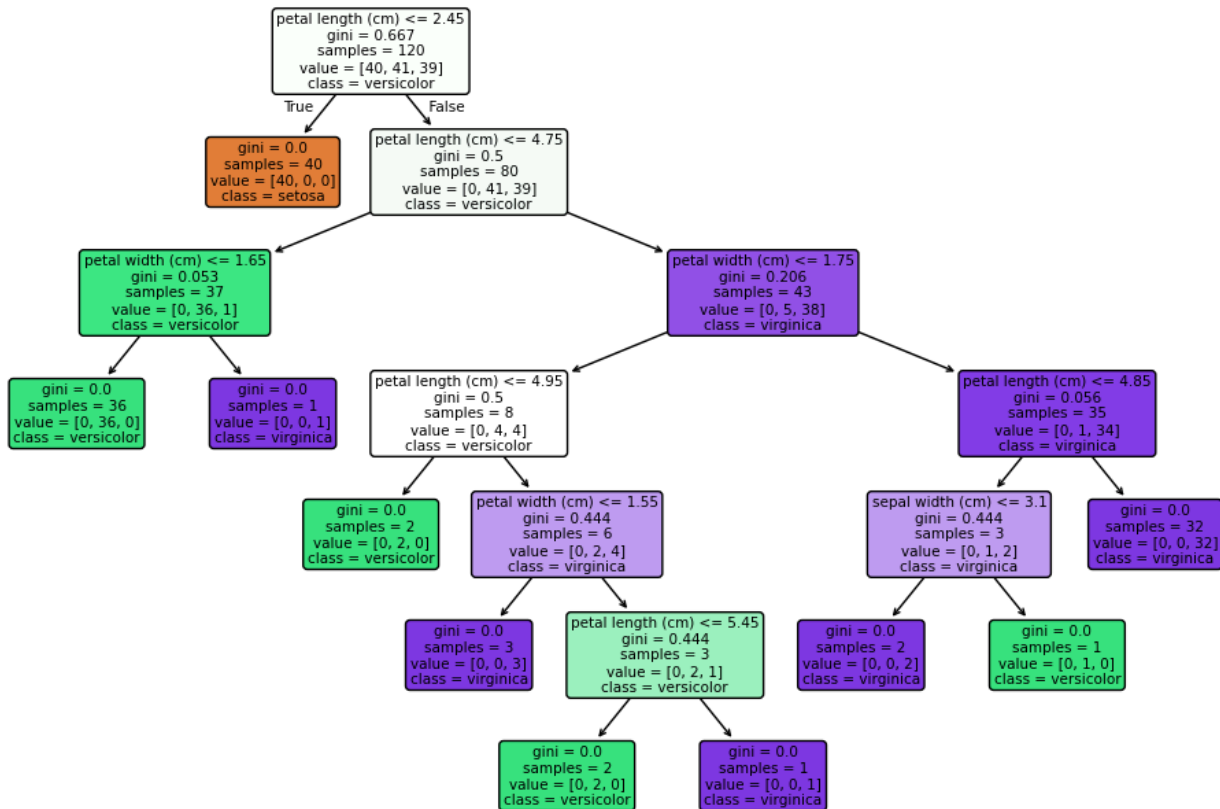
### ▼ 5. Visualising the decision tree

To make our model interpretable, we visualise the decision tree. The `filled=True` parameter colours the nodes based on the predicted class, and we label the nodes with feature names and class names. This makes it easier to understand how decisions are made based on feature values.

```
plt.figure(figsize=(12,8))
plot_tree(clf_iris, filled=True, feature_names=iris.feature_names, class_names=iris.target_names, rounded=True)
plt.title('Decision Tree Visualisation')
plt.show()
```



Decision Tree Visualisation



## 6. Evaluating the model

Finally, we evaluate the performance of our model. We use the `predict()` method on the test set (`x_test`) to generate predictions, and then calculate the accuracy of the model by comparing the predicted values (`y_pred`) to the actual test values (`y_test`).

```
y_pred = clf_iris.predict(X_test)

accuracy = (y_pred == y_test).mean()
print(f'Accuracy of the decision tree model: {accuracy:.2f}')
```



Accuracy of the decision tree model: 1.00