DMML-Lecture05-23jan2024

January 23, 2024

Chapter 6 – Decision Trees

This notebook contains all the sample code and solutions to the exercises in chapter 6.

1 Setup

This project requires Python 3.7 or above:

[1]:
```
import sys

assert sys.version_info >= (3, 7)
```

It also requires Scikit-Learn 1.0.1:

[2]:
```
from packaging import version
import sklearn

assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

As we did in previous chapters, let's define the default font sizes to make the figures prettier:

[3]:
```
import matplotlib.pyplot as plt

plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)
```

And let's create the `images/decision_trees` folder (if it doesn't already exist), and define the `save_fig()` function which is used through this notebook to save the figures in high-res for the book:

[4]:
```
from pathlib import Path

IMAGES_PATH = Path() / "images" / "decision_trees"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
```

```
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

2 Training and Visualizing a Decision Tree

[5]:
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

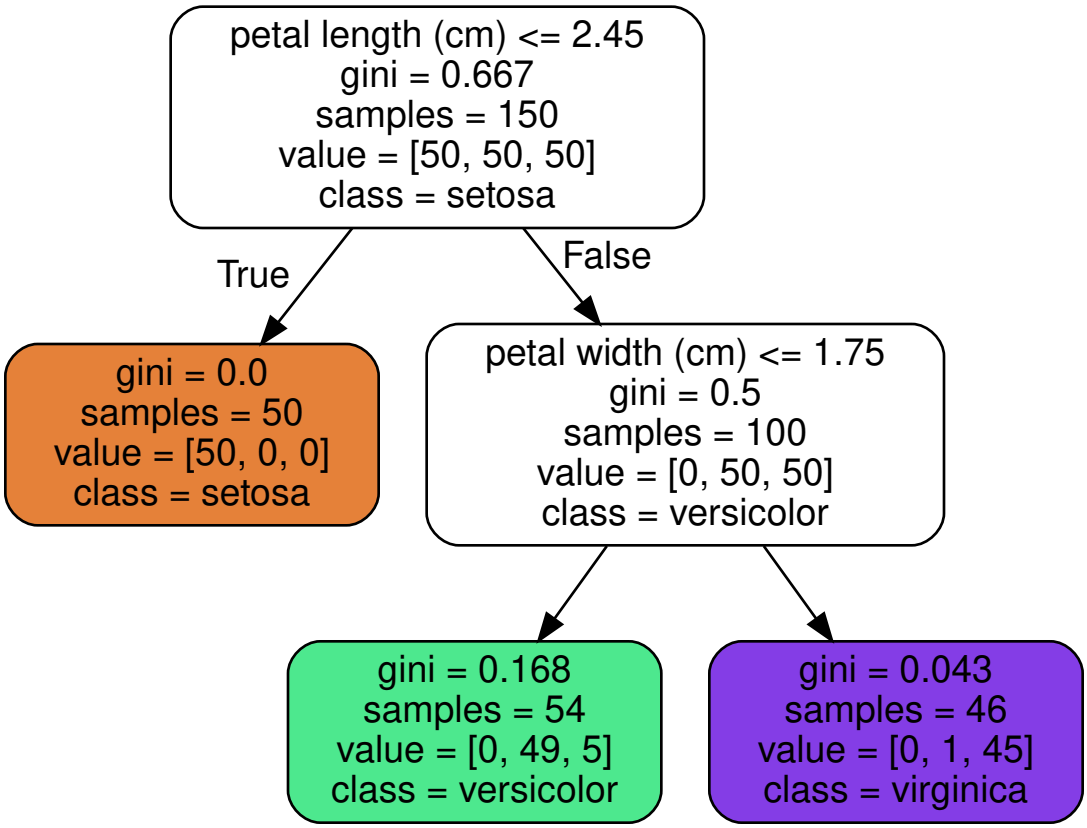
[5]:
```
DecisionTreeClassifier(max_depth=2, random_state=42)
```

This code example generates Figure 6–1. Iris Decision Tree:

[6]:
```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=str(IMAGES_PATH / "iris_tree.dot"),  # path differs in the book
        feature_names=["petal length (cm)", "petal width (cm)"],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

[7]:
```
from graphviz import Source

Source.from_file(IMAGES_PATH / "iris_tree.dot")  # path differs in the book
```

[7]:



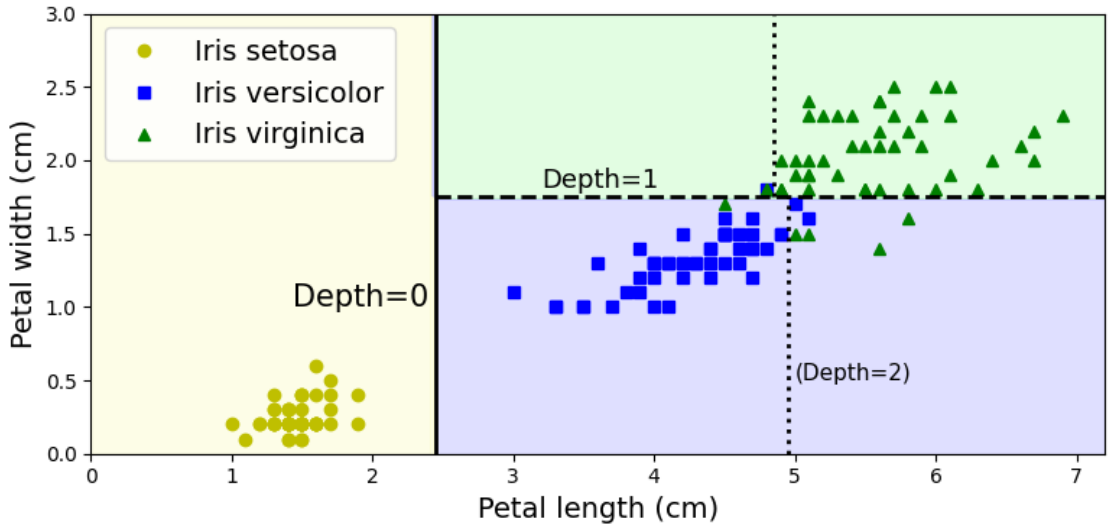
Graphviz also provides the `dot` command line tool to convert `.dot` files to a variety of formats. The following command converts the dot file to a png image:

[8]:
```
# extra code
!dot -Tpng {IMAGES_PATH / "iris_tree.dot"} -o {IMAGES_PATH / "iris_tree.png"}
```

3 Making Predictions

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

# extra code - just formatting details
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0', '#9898ff', '#a0faa0'])
plt.figure(figsize=(8, 4))

lengths, widths = np.meshgrid(np.linspace(0, 7.2, 100), np.linspace(0, 3, 100))
X_iris_all = np.c_[lengths.ravel(), widths.ravel()]
y_pred = tree_clf.predict(X_iris_all).reshape(lengths.shape)
plt.contourf(lengths, widths, y_pred, alpha=0.3, cmap=custom_cmap)
```

```
for idx, (name, style) in enumerate(zip(iris.target_names, ("yo", "bs", "g^"))):
    plt.plot(X_iris[:, 0][y_iris == idx], X_iris[:, 1][y_iris == idx],
             style, label=f"Iris {name}")

# extra code - this section beautifies and saves Figure 6-2
tree_clf_deeper = DecisionTreeClassifier(max_depth=3, random_state=42)
tree_clf_deeper.fit(X_iris, y_iris)
th0, th1, th2a, th2b = tree_clf_deeper.tree_.threshold[[0, 2, 3, 6]]
plt.xlabel("Petal length (cm)")
plt.ylabel("Petal width (cm)")
plt.plot([th0, th0], [0, 3], "k-", linewidth=2)
plt.plot([th0, 7.2], [th1, th1], "k--", linewidth=2)
plt.plot([th2a, th2a], [0, th1], "k:", linewidth=2)
plt.plot([th2b, th2b], [th1, 3], "k:", linewidth=2)
plt.text(th0 - 0.05, 1.0, "Depth=0", horizontalalignment="right", fontsize=15)
plt.text(3.2, th1 + 0.02, "Depth=1", verticalalignment="bottom", fontsize=13)
plt.text(th2a + 0.05, 0.5, "(Depth=2)", fontsize=11)
plt.axis([0, 7.2, 0, 3])
plt.legend()
save_fig("decision_tree_decision_boundaries_plot")

plt.show()
```



You can access the tree structure via the `tree_` attribute:

[10]:
```
tree_clf.tree_
```

[10]:
```
<sklearn.tree._tree.Tree at 0x7fc39010e090>
```

For more information, check out this class's documentation:

[11]:
```
# help(sklearn.tree._tree.Tree)
```

See the extra material section below for an example.

4 Estimating Class Probabilities

[12]:
```
tree_clf.predict_proba([[5, 1.5]]).round(3)
```

[12]:
```
array([[0.   , 0.907, 0.093]])
```

[13]:
```
tree_clf.predict([[5, 1.5]])
```

[13]:
```
array([1])
```

[14]:
```
49/54, 5/54
```

[14]:
```
(0.9074074074074074, 0.09259259259259259)
```

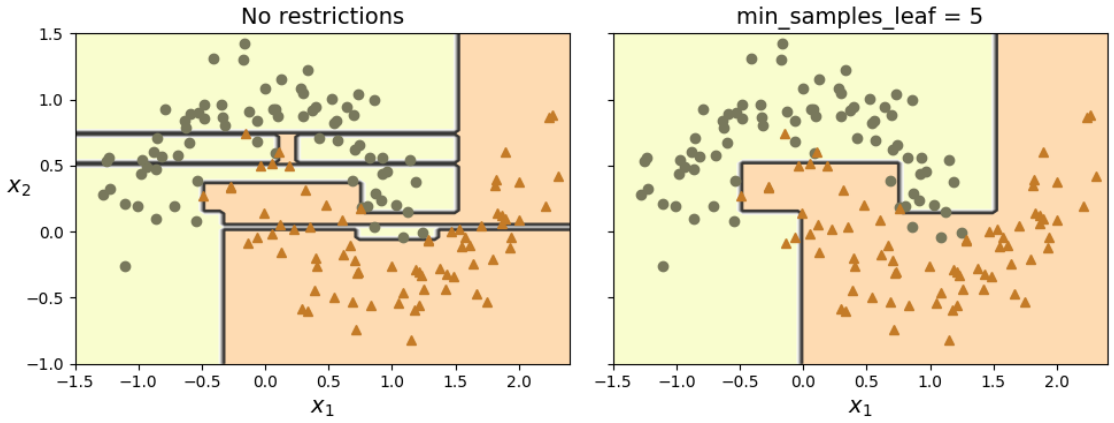
5 Regularization Hyperparameters

[15]:
```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

[15]:
```
DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
```

[16]:
```
# extra code - this cell generates and saves Figure 6-3

def plot_decision_boundary(clf, X, y, axes, cmap):
    x1, x2 = np.meshgrid(np.linspace(axes[0], axes[1], 100),
                         np.linspace(axes[2], axes[3], 100))
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)

    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=cmap)
    plt.contour(x1, x2, y_pred, cmap="Greys", alpha=0.8)
    colors = {"Wistia": ["#78785c", "#c47b27"], "Pastel1": ["red", "blue"]}
    markers = ("o", "^")
    for idx in (0, 1):
        plt.plot(X[:, 0][y == idx], X[:, 1][y == idx],
                 color=colors[cmap][idx], marker=markers[idx], linestyle="none")
    plt.axis(axes)
```

```
    plt.xlabel(r"$x_1$")
    plt.ylabel(r"$x_2$", rotation=0)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(axes[0])
plot_decision_boundary(tree_clf1, X_moons, y_moons,
                       axes=[-1.5, 2.4, -1, 1.5], cmap="Wistia")
plt.title("No restrictions")
plt.sca(axes[1])
plot_decision_boundary(tree_clf2, X_moons, y_moons,
                       axes=[-1.5, 2.4, -1, 1.5], cmap="Wistia")
plt.title(f"min_samples_leaf = {tree_clf2.min_samples_leaf}")
plt.ylabel("")
save_fig("min_samples_leaf_plot")
plt.show()
```



[17]:
```
X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2,
                                        random_state=43)
tree_clf1.score(X_moons_test, y_moons_test)
```

[17]: 0.898

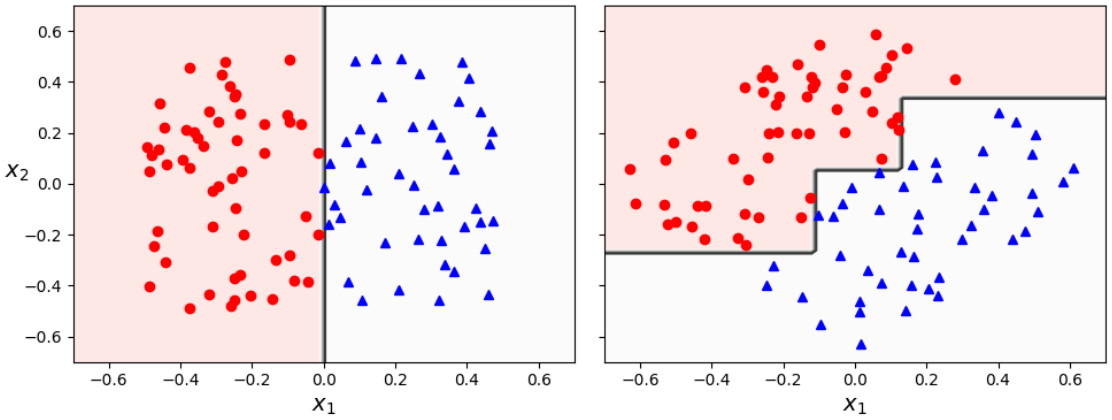
[18]:
```
tree_clf2.score(X_moons_test, y_moons_test)
```

[18]: 0.92

6 Sensitivity to axis orientation

Rotating the dataset also leads to completely different decision boundaries:

[19]:
```
# extra code - this cell generates and saves Figure 6-7

np.random.seed(6)
X_square = np.random.rand(100, 2) - 0.5
y_square = (X_square[:, 0] > 0).astype(np.int64)

angle = np.pi / 4  # 45 degrees
rotation_matrix = np.array([[np.cos(angle), -np.sin(angle)],
                            [np.sin(angle), np.cos(angle)]])
X_rotated_square = X_square.dot(rotation_matrix)

tree_clf_square = DecisionTreeClassifier(random_state=42)
tree_clf_square.fit(X_square, y_square)
tree_clf_rotated_square = DecisionTreeClassifier(random_state=42)
tree_clf_rotated_square.fit(X_rotated_square, y_square)

fig, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
plt.sca(axes[0])
plot_decision_boundary(tree_clf_square, X_square, y_square,
                       axes=[-0.7, 0.7, -0.7, 0.7], cmap="Pastel1")
plt.sca(axes[1])
plot_decision_boundary(tree_clf_rotated_square, X_rotated_square, y_square,
                       axes=[-0.7, 0.7, -0.7, 0.7], cmap="Pastel1")
plt.ylabel("")

save_fig("sensitivity_to_rotation_plot")
plt.show()
```

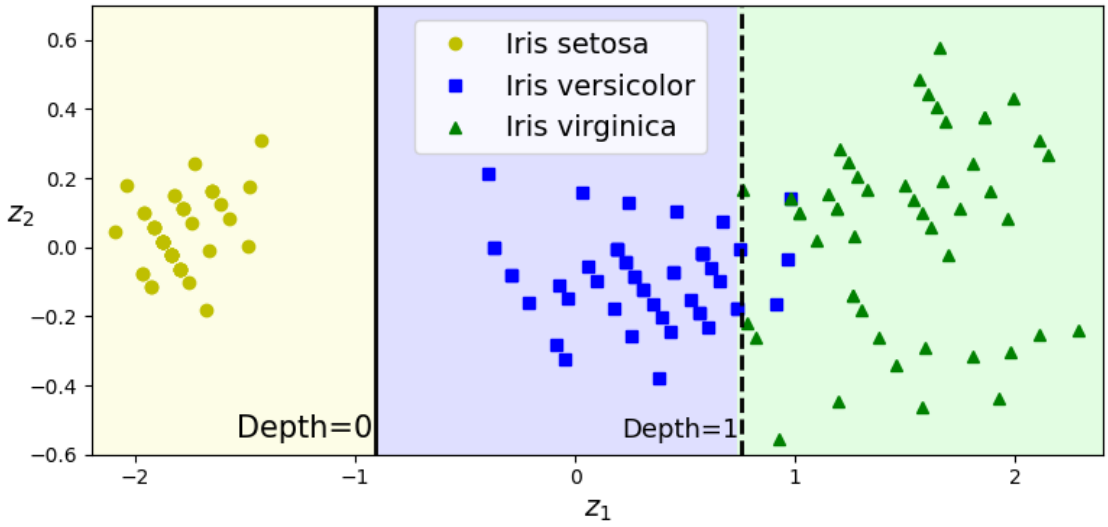


[20]:
```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

```
pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

[20]:
```
DecisionTreeClassifier(max_depth=2, random_state=42)
```

[21]:
```
# extra code - this cell generates and saves Figure 6-8

plt.figure(figsize=(8, 4))

axes = [-2.2, 2.4, -0.6, 0.7]
z0s, z1s = np.meshgrid(np.linspace(axes[0], axes[1], 100),
                       np.linspace(axes[2], axes[3], 100))
X_iris_pca_all = np.c_[z0s.ravel(), z1s.ravel()]
y_pred = tree_clf_pca.predict(X_iris_pca_all).reshape(z0s.shape)

plt.contourf(z0s, z1s, y_pred, alpha=0.3, cmap=custom_cmap)
for idx, (name, style) in enumerate(zip(iris.target_names, ("yo", "bs", "g^"))):
    plt.plot(X_iris_rotated[:, 0][y_iris == idx],
             X_iris_rotated[:, 1][y_iris == idx],
             style, label=f"Iris {name}")

plt.xlabel("$z_1$")
plt.ylabel("$z_2$", rotation=0)
th1, th2 = tree_clf_pca.tree_.threshold[[0, 2]]
plt.plot([th1, th1], axes[2:], "k-", linewidth=2)
plt.plot([th2, th2], axes[2:], "k--", linewidth=2)
plt.text(th1 - 0.01, axes[2] + 0.05, "Depth=0",
         horizontalalignment="right", fontsize=15)
plt.text(th2 - 0.01, axes[2] + 0.05, "Depth=1",
         horizontalalignment="right", fontsize=13)
plt.axis(axes)
plt.legend(loc=(0.32, 0.67))
save_fig("pca_preprocessing_plot")

plt.show()
```



7 Decision Trees Have High Variance

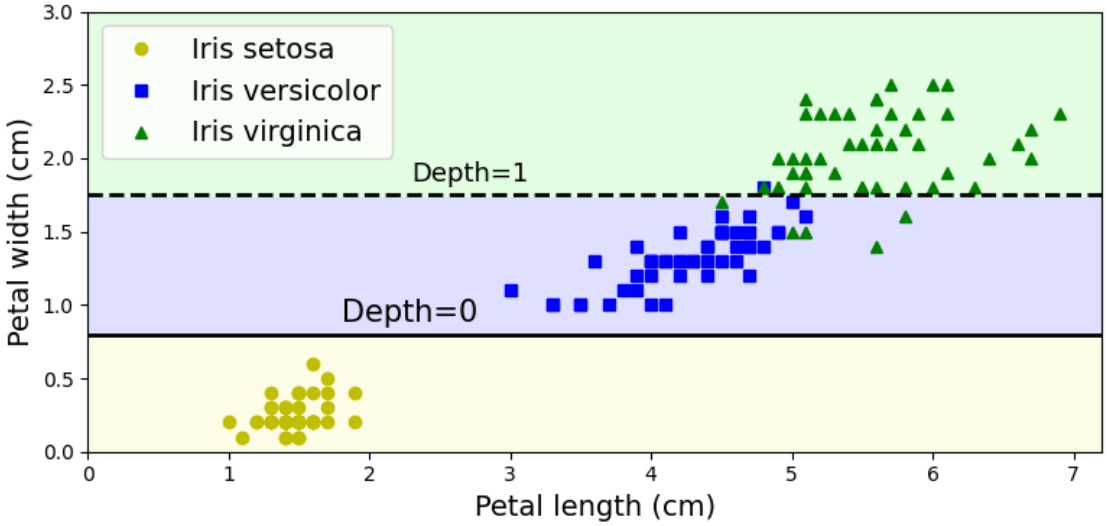
We've seen that small changes in the dataset (such as a rotation) may produce a very different Decision Tree. Now let's show that training the same model on the same data may produce a very different model every time, since the CART training algorithm used by Scikit-Learn is stochastic. To show this, we will set `random_state` to a different value than earlier:

[22]:
```
tree_clf_tweaked = DecisionTreeClassifier(max_depth=2, random_state=40)
tree_clf_tweaked.fit(X_iris, y_iris)
```

[22]:
```
DecisionTreeClassifier(max_depth=2, random_state=40)
```

[23]:
```
# extra code - this cell generates and saves Figure 6-9

plt.figure(figsize=(8, 4))
y_pred = tree_clf_tweaked.predict(X_iris_all).reshape(lengths.shape)
plt.contourf(lengths, widths, y_pred, alpha=0.3, cmap=custom_cmap)

for idx, (name, style) in enumerate(zip(iris.target_names, ("yo", "bs", "g^"))):
    plt.plot(X_iris[:, 0][y_iris == idx], X_iris[:, 1][y_iris == idx],
             style, label=f"Iris {name}")

th0, th1 = tree_clf_tweaked.tree_.threshold[[0, 2]]
plt.plot([0, 7.2], [th0, th0], "k-", linewidth=2)
plt.plot([0, 7.2], [th1, th1], "k--", linewidth=2)
plt.text(1.8, th0 + 0.05, "Depth=0", verticalalignment="bottom", fontsize=15)
plt.text(2.3, th1 + 0.05, "Depth=1", verticalalignment="bottom", fontsize=13)
plt.xlabel("Petal length (cm)")
```

```
plt.ylabel("Petal width (cm)")
plt.axis([0, 7.2, 0, 3])
plt.legend()
save_fig("decision_tree_high_variance_plot")

plt.show()
```



8 Extra Material – Accessing the tree structure

A trained `DecisionTreeClassifier` has a `tree_` attribute that stores the tree's structure:

[24]:
```
tree = tree_clf.tree_
tree
```

[24]:
```
<sklearn.tree._tree.Tree at 0x7fc39010e090>
```

You can get the total number of nodes in the tree:

[25]:
```
tree.node_count
```

[25]:
```
5
```

And other self-explanatory attributes are available:

[26]:
```
tree.max_depth
```

[26]:
```
2
```

[27]:
```
tree.max_n_classes
```

[27]: 3

[28]:
```
tree.n_features
```

[28]: 2

[29]:
```
tree.n_outputs
```

[29]: 1

[30]:
```
tree.n_leaves
```

[30]: 3

All the information about the nodes is stored in NumPy arrays. For example, the impurity of each node:

[31]:
```
tree.impurity
```

[31]: array([0.66666667, 0.   , 0.5   , 0.16803841, 0.04253308])

The root node is at index 0. The left and right children nodes of node i are `tree.children_left[i]` and `tree.children_right[i]`. For example, the children of the root node are:

[32]:
```
tree.children_left[0], tree.children_right[0]
```

[32]: (1, 2)

When the left and right nodes are equal, it means this is a leaf node (and the children node ids are arbitrary):

[33]:
```
tree.children_left[3], tree.children_right[3]
```

[33]: (-1, -1)

So you can get the leaf node ids like this:

[34]:
```
is_leaf = (tree.children_left == tree.children_right)
np.arange(tree.node_count)[is_leaf]
```

[34]: array([1, 3, 4])

Non-leaf nodes are called *split nodes*. The feature they split is available via the `feature` array. Values for leaf nodes should be ignored:

[35]:
```
tree.feature
```

[35]: array([0, -2, 1, -2, -2], dtype=int64)

And the corresponding thresholds are:

[36]:
```
tree.threshold
```

[36]:
```
array([ 2.44999999, -2.        ,  1.75      , -2.        , -2.        ])
```

And the number of instances per class that reached each node is available too:

[37]:
```
tree.value
```

[37]:
```
array([[[0.33333333, 0.33333333, 0.33333333]],

       [[1.        , 0.        , 0.        ]],

       [[0.        , 0.5       , 0.5       ]],

       [[0.        , 0.90740741, 0.09259259]],

       [[0.        , 0.02173913, 0.97826087]]])
```

[38]:
```
tree.n_node_samples
```

[38]:
```
array([150,  50, 100,  54,  46], dtype=int64)
```

[39]:
```
np.all(tree.value.sum(axis=(1, 2)) == tree.n_node_samples)
```

[39]:
```
False
```

Here's how you can compute the depth of each node:

[40]:
```
def compute_depth(tree_clf):
    tree = tree_clf.tree_
    depth = np.zeros(tree.node_count)
    stack = [(0, 0)]
    while stack:
        node, node_depth = stack.pop()
        depth[node] = node_depth
        if tree.children_left[node] != tree.children_right[node]:
            stack.append((tree.children_left[node], node_depth + 1))
            stack.append((tree.children_right[node], node_depth + 1))
    return depth

depth = compute_depth(tree_clf)
depth
```

[40]:
```
array([0., 1., 1., 2., 2.])
```

Here's how to get the thresholds of all split nodes at depth 1:

[41]:
```
tree_clf.tree_.feature[(depth == 1) & (~is_leaf)]
```

[41]:
```
array([1], dtype=int64)
```

[42]:
```
tree_clf.tree_.threshold[(depth == 1) & (~is_leaf)]
```

[42]:
```
array([1.75])
```