DMML Lecture 05, 28 Jan 2025 – Decision Trees

Setup

This project requires Python 3.7 or above:

In [1]:
```
import sys

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

It also requires Scikit-Learn ≥ 1.0.1:

In [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:

In [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:

In [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)
```

Training and Visualizing a Decision Tree

In [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)
```

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

Examine `iris` data

In [6]:
```
iris
```

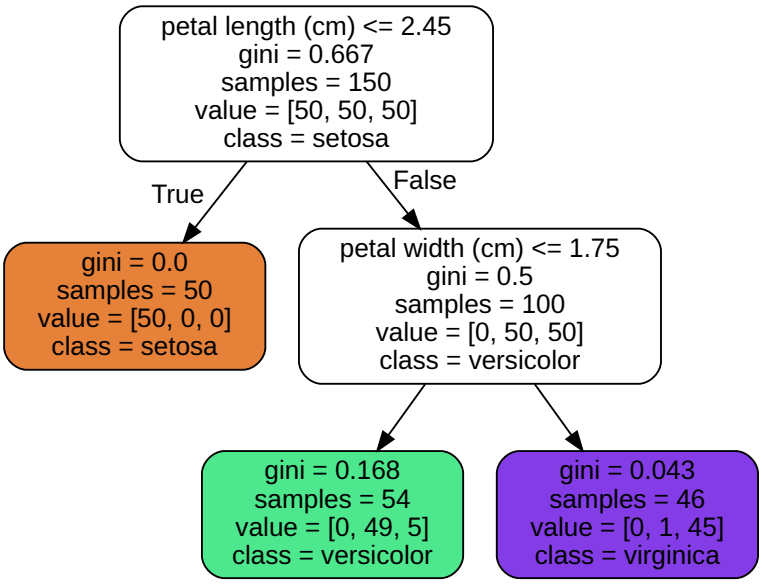
Out[6]:
```
{'data':      sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
 0                   5.1               3.5                1.4               0.2
 1                   4.9               3.0                1.4               0.2
 2                   4.7               3.2                1.3               0.2
 3                   4.6               3.1                1.5               0.2
 4                   5.0               3.6                1.4               0.2
 ..                  ...               ...                ...               ...
 145                 6.7               3.0                5.2               2.3
 146                 6.3               2.5                5.0               1.9
 147                 6.5               3.0                5.2               2.0
 148                 6.2               3.4                5.4               2.3
 149                 5.9               3.0                5.1               1.8

 [150 rows x 4 columns],
 'target': 0      0
 1      0
 2      0
 3      0
 4      0
       ..
 145    2
 146    2
 147    2
 148    2
 149    2
 Name: target, Length: 150, dtype: int64,
 'frame':      sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
 0                   5.1               3.5                1.4               0.2
 1                   4.9               3.0                1.4               0.2
 2                   4.7               3.2                1.3               0.2
 3                   4.6               3.1                1.5               0.2
 4                   5.0               3.6                1.4               0.2
 ..                  ...               ...                ...               ...
 145                 6.7               3.0                5.2               2.3
 146                 6.3               2.5                5.0               1.9
 147                 6.5               3.0                5.2               2.0
 148                 6.2               3.4                5.4               2.3
 149                 5.9               3.0                5.1               1.8

      target
 0         0
 1         0
 2         0
 3         0
 4         0
 ..      ...
 145       2
 146       2
 147       2
 148       2
 149       2

 [150 rows x 5 columns],
 'target_names': array(['setosa', 'versicolor', 'virginica'], dtype='<U10'),
 'DESCR': '.. _iris_dataset:\n\nIris plants dataset\n--------------------\n\n**Data Set Characteristics:**\n
\n:Number of Instances: 150 (50 in each of three classes)\n:Number of Attributes: 4 numeric, predictive attrib
utes and the class\n:Attribute Information:\n    - sepal length in cm\n    - sepal width in cm\n    - petal le
ngth in cm\n    - petal width in cm\n    - class:\n            - Iris-Setosa\n            - Iris-Versicolour\n
- Iris-Virginica\n\n:Summary Statistics:\n\n============== ==== ==== ======= ===== ====================\n
Min  Max   Mean    SD   Class Correlation\n============== ==== ==== ======= ===== ====================\nsepal
length:   4.3  7.9   5.84   0.83    0.7826\nsepal width:    2.0  4.4   3.05   0.43   -0.4194\npetal length:
1.0  6.9   3.76   1.76    0.9490  (high!)\npetal width:    0.1  2.5   1.20   0.76    0.9565  (high!)\n========
====== ==== ==== ======= ===== ====================\n\n:Missing Attribute Values: None\n:Class Distribution: 3
3.3% for each of 3 classes.\n:Creator: R.A. Fisher\n:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)\n:
Date: July, 1988\n\nThe famous Iris database, first used by Sir R.A. Fisher. The dataset is taken\nfrom Fisher
\'s paper. Note that it\'s the same as in R, but not as in the UCI\nMachine Learning Repository, which has two
wrong data points.\n\nThis is perhaps the best known database to be found in the\npattern recognition literatu
re.  Fisher\'s paper is a classic in the field and\nis referenced frequently to this day.  (See Duda & Hart, f
or example.)  The\ndata set contains 3 classes of 50 instances each, where each class refers to a\ntype of iri
s plant.  One class is linearly separable from the other 2; the\nlatter are NOT linearly separable from each o
ther.\n\n.. dropdown:: References\n\n  - Fisher, R.A. "The use of multiple measurements in taxonomic problem
s"\n    Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to\n    Mathematical Statistics"
(John Wiley, NY, 1950).\n  - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.\n
(Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.\n  - Dasarathy, B.V. (1980) "Nosing Around t
he Neighborhood: A New System\n    Structure and Classification Rule for Recognition in Partially Exposed\n
Environments".  IEEE Transactions on Pattern Analysis and Machine\n    Intelligence, Vol. PAMI-2, No. 1, 67-7
1.\n  - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions\n    on Information Theory,
May 1972, 431-433.\n  - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II\n    conceptual
clustering system finds 3 classes in the data.\n  - Many, many more ...\n',
 'feature_names': ['sepal length (cm)',
  'sepal width (cm)',
  'petal length (cm)',
  'petal width (cm)'],
 'filename': 'iris.csv',
 'data_module': 'sklearn.datasets.data'}
```

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

In [7]:
```
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
    )
```

In [8]:
```
from graphviz import Source

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

Out[8]:



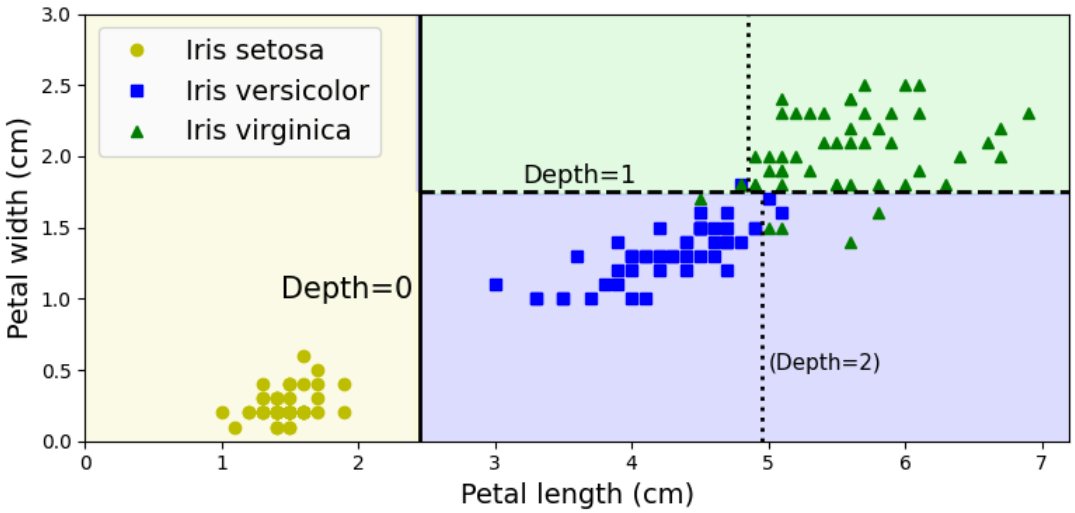
Graphviz also provides the `dot` command line tool to convert `.dot` files to a variety of formats. The following command converts the dot file to a png image:

In [9]:
```
# extra code
!dot -Tpng {IMAGES_PATH / "iris_tree.dot"} -o {IMAGES_PATH / "iris_tree.png"}
```

Making Predictions

In [10]:
```
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:

In [11]:
```
tree_clf.tree_
```

Out[11]:
```
<sklearn.tree._tree.Tree at 0x7fa76efd5140>
```

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

In [12]:
```
# help(sklearn.tree._tree.Tree)
```

See the extra material section below for an example.

Estimating Class Probabilities

In [13]:
```
tree_clf.predict_proba([[5, 1.5]]).round(3)
```

Out[13]:
```
array([[0.   , 0.907, 0.093]])
```

In [14]:
```
tree_clf.predict([[5, 1.5]])
```

Out[14]:
```
array([1])
```

In [15]:
```
tree_clf.predict_proba([[5, 2.5]]).round(3)
```

Out[15]:
```
array([[0.   , 0.022, 0.978]])
```

In [16]:
```
tree_clf.predict([[5, 2.5]])
```

Out[16]:
```
array([2])
```

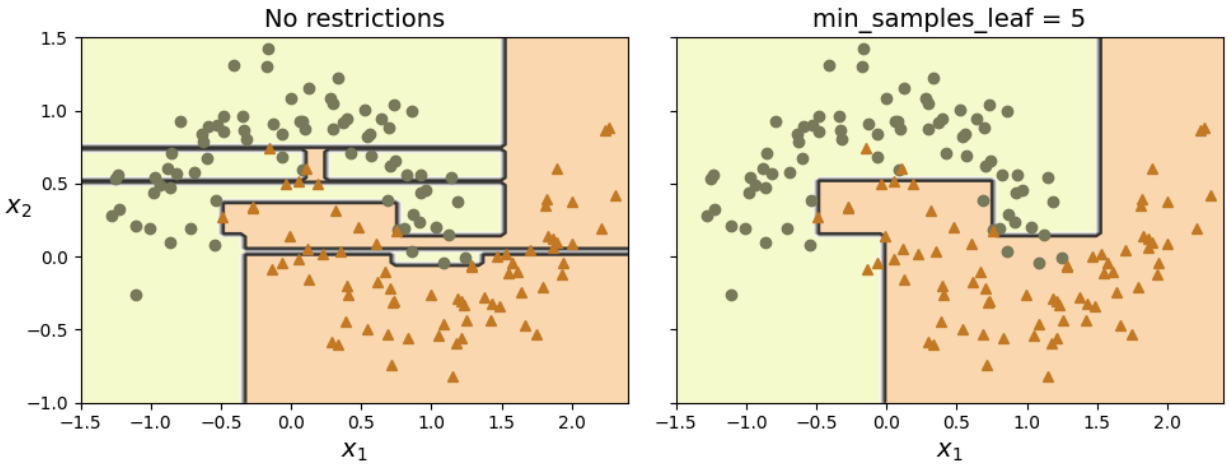
Regularization Hyperparameters

In [17]:
```
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)
```

Out[17]:
```
DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
```

In [18]:
```
# 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()
```



In [19]:
```
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)
```

Out[19]: 0.898

In [20]:
```
tree_clf2.score(X_moons_test, y_moons_test)
```

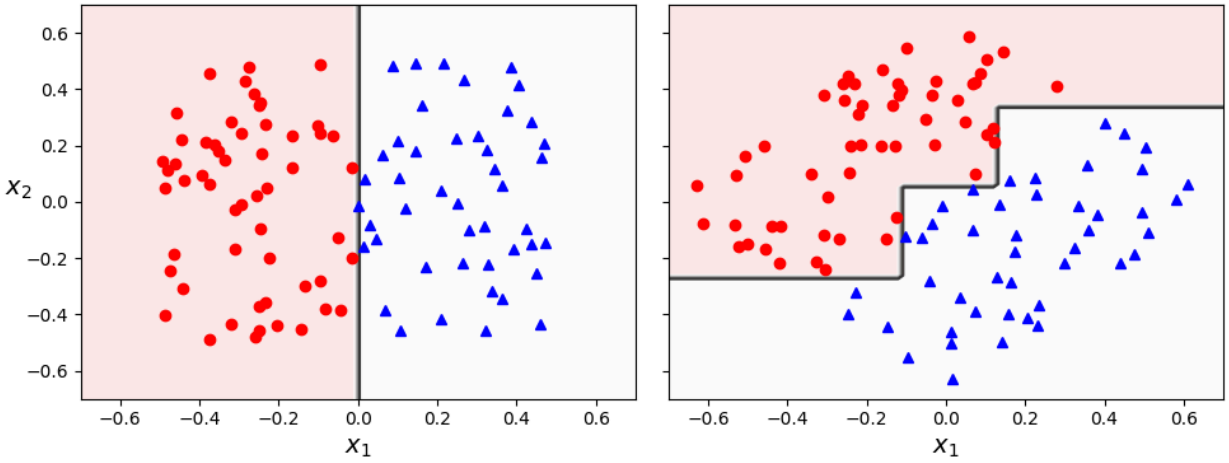
Out[20]: 0.92

Sensitivity to axis orientation

Rotating the dataset also leads to completely different decision boundaries:

In [21]:
```
# 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()
```



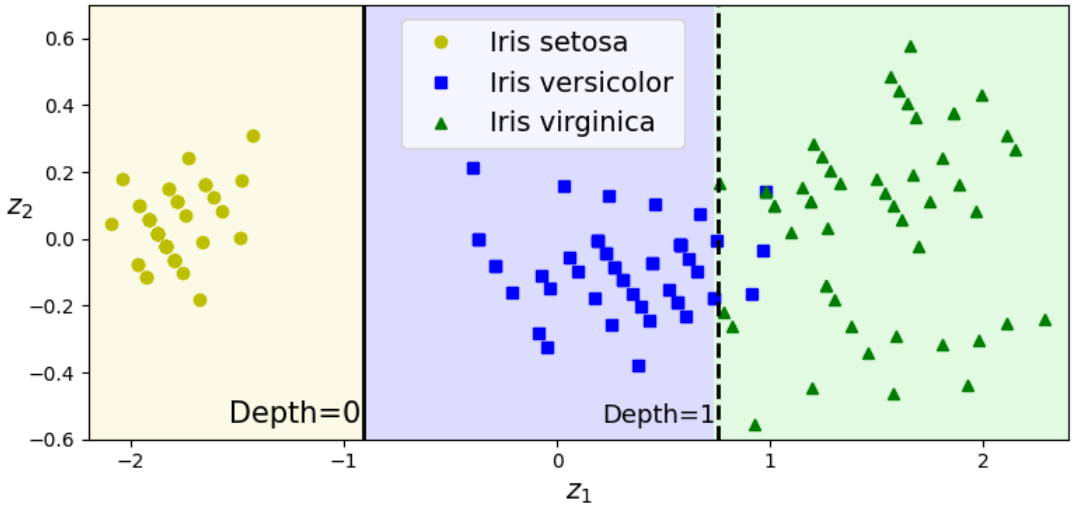
In [22]:
```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

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)
```

Out[22]:

DecisionTreeClassifier

DecisionTreeClassifier(max_depth=2, random_state=42)

In [23]:
```
# 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()
```



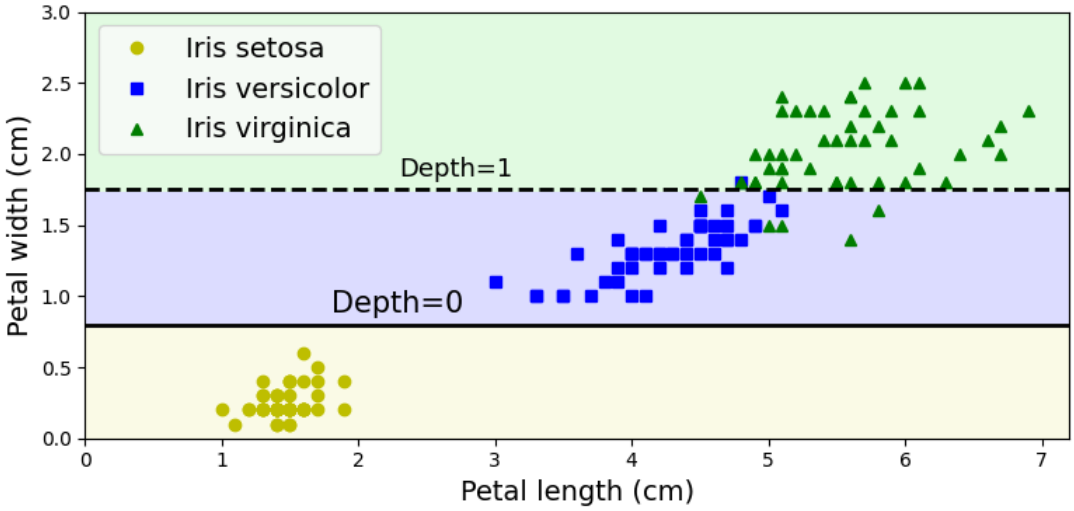
Decision Trees Have High Variance

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:

In [24]:
```
tree_clf_tweaked = DecisionTreeClassifier(max_depth=2, random_state=40)
tree_clf_tweaked.fit(X_iris, y_iris)
```

Out[24]:
```
DecisionTreeClassifier(max_depth=2, random_state=40)
```

In [25]:
```
# 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()
```



Extra Material – Accessing the tree structure

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

In [26]:
```
tree = tree_clf.tree_
tree
```

Out[26]:
```
<sklearn.tree._tree.Tree at 0x7fa76efd5140>
```

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

In [27]:
```
tree.node_count
```

Out[27]:
```
5
```

And other self-explanatory attributes are available:

In [28]:
```
tree.max_depth
```

Out[28]:
```
2
```

In [29]:
```
tree.max_n_classes
```

Out[29]:
```
3
```

In [30]:
```
tree.n_features
```

Out[30]:
```
2
```

In [31]:
```
tree.n_outputs
```

Out[31]:
```
1
```

In [32]:
```
tree.n_leaves
```

Out[32]:
```
np.int64(3)
```

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

In [33]:
```
tree.impurity
```

Out[33]:
```
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:

In [34]:
```
tree.children_left[0], tree.children_right[0]
```

Out[34]:
```
(np.int64(1), np.int64(2))
```

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

In [35]:
```
tree.children_left[3], tree.children_right[3]
```

Out[35]:
```
(np.int64(-1), np.int64(-1))
```

So you can get the leaf node ids like this:

In [36]:
```
is_leaf = (tree.children_left == tree.children_right)
np.arange(tree.node_count)[is_leaf]
```

Out[36]:
```
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:

In [37]:
```
tree.feature
```

Out[37]:
```
array([ 0, -2,  1, -2, -2], dtype=int64)
```

And the corresponding thresholds are:

In [38]:
```
tree.threshold
```

Out[38]:
```
array([ 2.44999999, -2.        ,  1.75      , -2.        , -2.        ])
```

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

In [39]:
```
tree.value
```

Out[39]:
```
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]]])
```

In [40]:
```
tree.n_node_samples
```

Out[40]:
```
array([150,  50, 100,  54,  46], dtype=int64)
```

In [41]:
```
np.all(tree.value.sum(axis=(1, 2)) == tree.n_node_samples)
```

Out[41]:
```
np.False_
```

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

In [42]:
```
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
```

Out[42]:
```
array([0., 1., 1., 2., 2.])
```

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

In [43]:
```
tree_clf.tree_.feature[(depth == 1) & (~is_leaf)]
```

Out[43]:
```
array([1], dtype=int64)
```

In [44]:
```
tree_clf.tree_.threshold[(depth == 1) & (~is_leaf)]
```

Out[44]:
```
array([1.75])
```