trees_hyperparameters

May 29, 2021

1 Importance of decision tree hyperparameters on generalization

In this notebook, we will illustrate the importance of some key hyperparameters on the decision tree; we will demonstrate their effects on the classification and regression problems we saw previously.

First, we will load the classification and regression datasets.

[1]:
```python
import pandas as pd

data_clf_columns = ["Culmen Length (mm)", "Culmen Depth (mm)"]
target_clf_column = "Species"
data_clf = pd.read_csv("../datasets/penguins_classification.csv")
```

[2]:
```python
data_reg_columns = ["Flipper Length (mm)"]
target_reg_column = "Body Mass (g)"
data_reg = pd.read_csv("../datasets/penguins_regression.csv")
```

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

1.1 Create helper functions

We will create two functions that will:

- fit a decision tree on some training data;
- show the decision function of the model.

[3]:
```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


def plot_classification(model, X, y, ax=None):
    from sklearn.preprocessing import LabelEncoder
    model.fit(X, y)

    range_features = {
        feature_name: (X[feature_name].min() - 1, X[feature_name].max() + 1)
```

```python
        for feature_name in X.columns
    }
    feature_names = list(range_features.keys())
    # create a grid to evaluate all possible samples
    plot_step = 0.02
    xx, yy = np.meshgrid(
        np.arange(*range_features[feature_names[0]], plot_step),
        np.arange(*range_features[feature_names[1]], plot_step),
    )

    # compute the associated prediction
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = LabelEncoder().fit_transform(Z)
    Z = Z.reshape(xx.shape)

    # make the plot of the boundary and the data samples
    if ax is None:
        _, ax = plt.subplots()
    ax.contourf(xx, yy, Z, alpha=0.4, cmap="RdBu")
    if y.nunique() == 3:
        palette = ["tab:red", "tab:blue", "black"]
    else:
        palette = ["tab:red", "tab:blue"]
    sns.scatterplot(
        x=data_clf_columns[0], y=data_clf_columns[1], hue=target_clf_column,
        data=data_clf, ax=ax, palette=palette)

    return ax
```

[4]:
```python
def plot_regression(model, X, y, ax=None):
    model.fit(X, y)

    X_test = pd.DataFrame(
        np.arange(X.iloc[:, 0].min(), X.iloc[:, 0].max()),
        columns=X.columns,
    )
    y_pred = model.predict(X_test)

    if ax is None:
        _, ax = plt.subplots()
    sns.scatterplot(x=X.iloc[:, 0], y=y, color="black", alpha=0.5, ax=ax)
    ax.plot(X_test, y_pred, linewidth=4)

    return ax
```

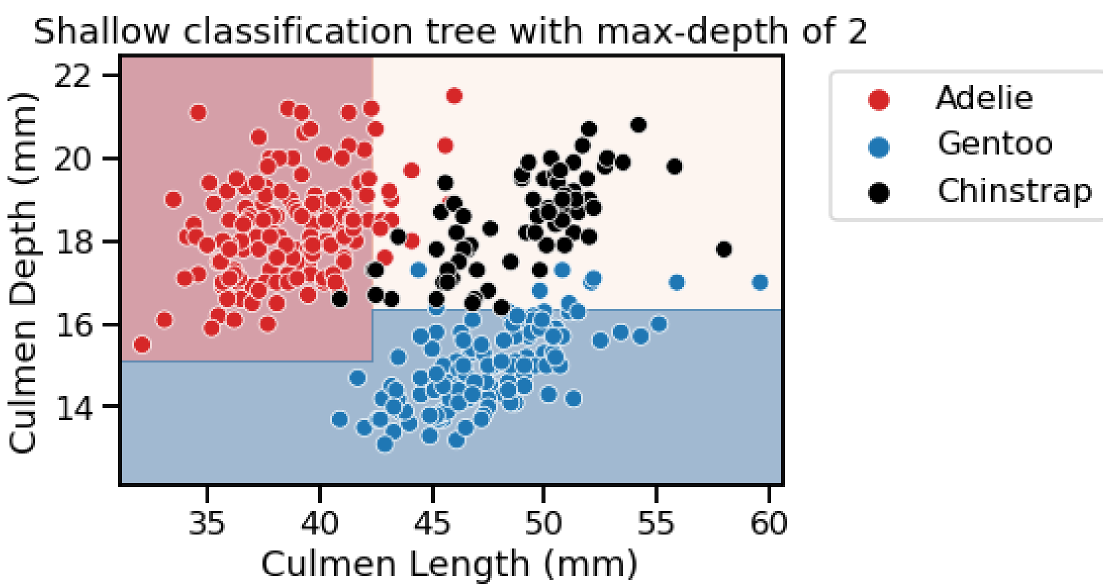
1.2 Effect of the max_depth parameter

The hyperparameter `max_depth` controls the overall complexity of a decision tree. This hyperparameter allows to get a trade-off between an under-fitted and over-fitted decision tree. Let's build a shallow tree and then a deeper tree, for both classification and regression, to understand the impact of the parameter.

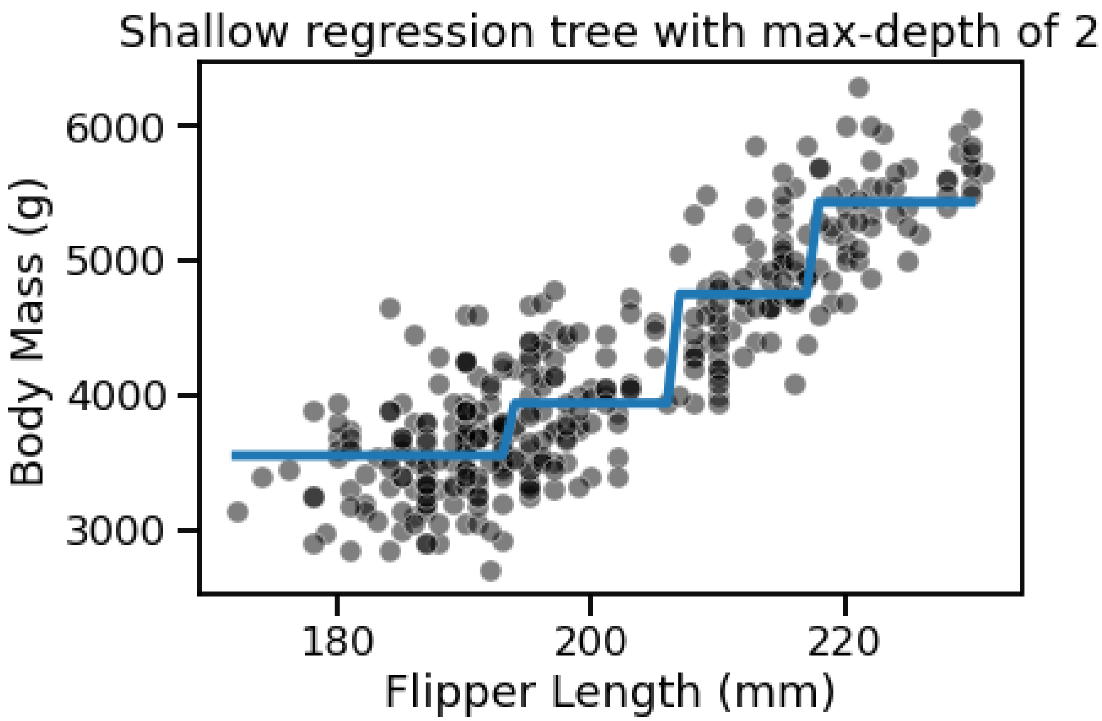
We can first set the `max_depth` parameter value to a very low value.

[5]:
```
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

max_depth = 2
tree_clf = DecisionTreeClassifier(max_depth=max_depth)
tree_reg = DecisionTreeRegressor(max_depth=max_depth)
```

[6]:
```
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
_ = plt.title(f"Shallow classification tree with max-depth of {max_depth}")
```



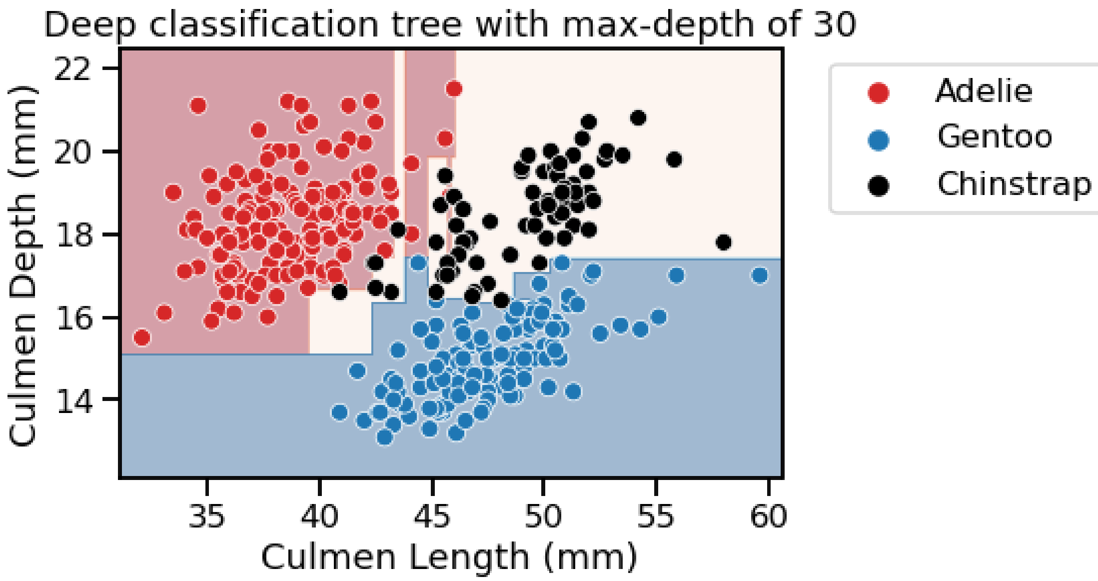
[7]:
```
plot_regression(tree_reg, data_reg[data_reg_columns],
                data_reg[target_reg_column])
_ = plt.title(f"Shallow regression tree with max-depth of {max_depth}")
```



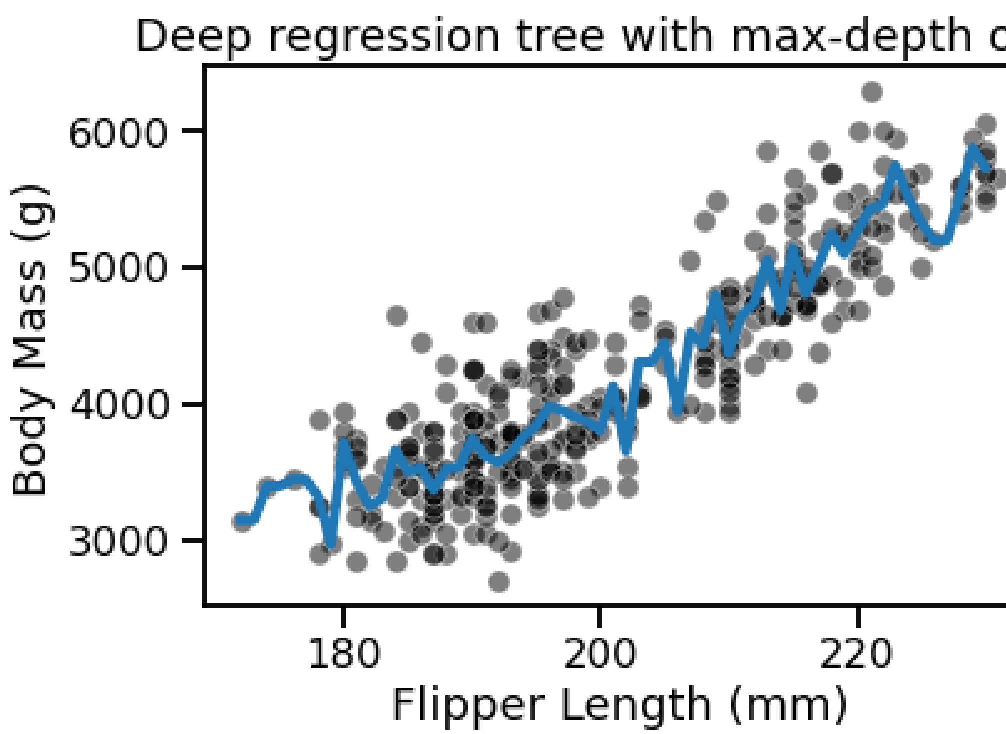
Now, let's increase the `max_depth` parameter value to check the difference by observing the decision function.

[8]:
```
max_depth = 30
tree_clf = DecisionTreeClassifier(max_depth=max_depth)
tree_reg = DecisionTreeRegressor(max_depth=max_depth)
```

[9]:
```
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
_ = plt.title(f"Deep classification tree with max-depth of {max_depth}")
```



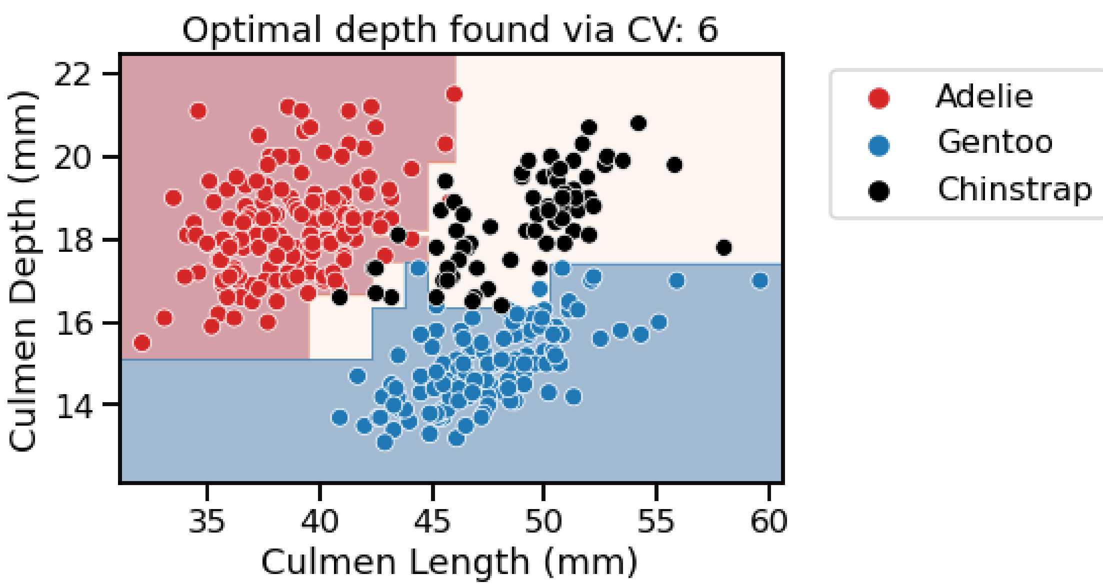
[10]:
```
plot_regression(tree_reg, data_reg[data_reg_columns],
                data_reg[target_reg_column])
_ = plt.title(f"Deep regression tree with max-depth of {max_depth}")
```



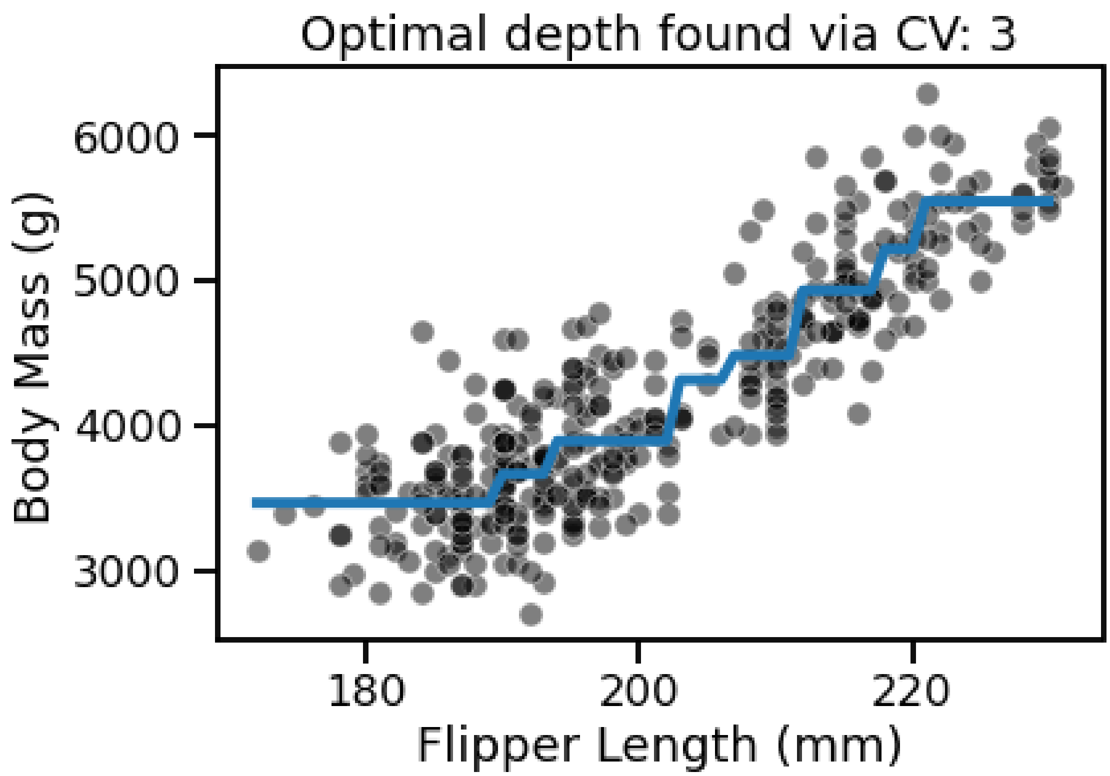
For both classification and regression setting, we observe that increasing the depth will make the tree model more expressive. However, a tree that is too deep will overfit the training data, creating partitions which are only correct for "outliers" (noisy samples). The `max_depth` is one of the hyperparameters that one should optimize via cross-validation and grid-search.

[11]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {"max_depth": np.arange(2, 10, 1)}
tree_clf = GridSearchCV(DecisionTreeClassifier(), param_grid=param_grid)
tree_reg = GridSearchCV(DecisionTreeRegressor(), param_grid=param_grid)
```

[12]:
```
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
_ = plt.title(f"Optimal depth found via CV: "
              f"{tree_clf.best_params_['max_depth']}")
```



[13]:
```
plot_regression(tree_reg, data_reg[data_reg_columns],
                data_reg[target_reg_column])
_ = plt.title(f"Optimal depth found via CV: "
              f"{tree_reg.best_params_['max_depth']}")
```



With this example, we see that there is not a single value that is optimal for any dataset. Thus, this parameter is required to be optimized for each application.

1.3 Other hyperparameters in decision trees

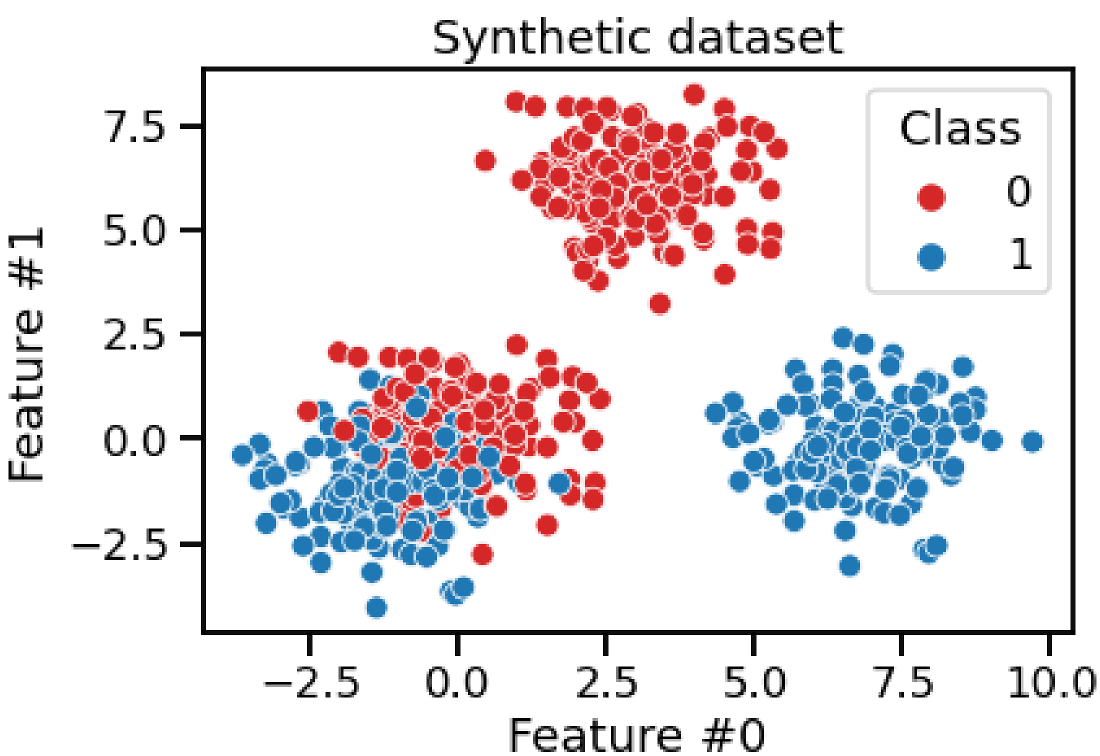
The `max_depth` hyperparameter controls the overall complexity of the tree. This parameter is adequate under the assumption that a tree is built is symmetric. However, there is not guarantee that a tree will be symmetric. Indeed, optimal statistical performance could be reached by growing some of the branches deeper than some others.

We will built a dataset where we will illustrate this asymmetry. We will generate a dataset composed of 2 subsets: one subset where a clear separation should be found by the tree and another subset where samples from both classes will be mixed. It implies that a decision tree will need more splits to classify properly samples from the second subset than from the first subset.

[14]:
```
from sklearn.datasets import make_blobs

data_clf_columns = ["Feature #0", "Feature #1"]
target_clf_column = "Class"

# Blobs that will be interlaced
X_1, y_1 = make_blobs(
    n_samples=300, centers=[[0, 0], [-1, -1]], random_state=0)
```

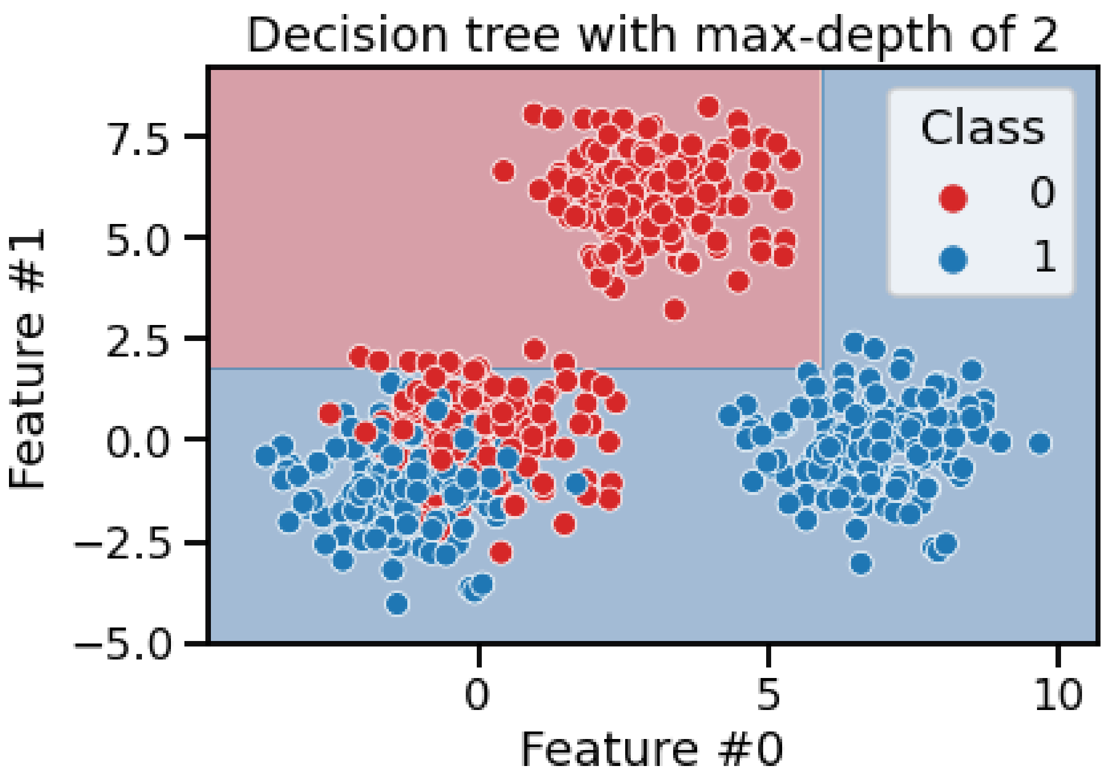
```python
# Blobs that will be easily separated
X_2, y_2 = make_blobs(
    n_samples=300, centers=[[3, 6], [7, 0]], random_state=0)

X = np.concatenate([X_1, X_2], axis=0)
y = np.concatenate([y_1, y_2])
data_clf = np.concatenate([X, y[:, np.newaxis]], axis=1)
data_clf = pd.DataFrame(
    data_clf, columns=data_clf_columns + [target_clf_column])
data_clf[target_clf_column] = data_clf[target_clf_column].astype(np.int32)
```

[15]:
```python
sns.scatterplot(data=data_clf, x=data_clf_columns[0], y=data_clf_columns[1],
                hue=target_clf_column, palette=["tab:red", "tab:blue"])
_ = plt.title("Synthetic dataset")
```



We will first train a shallow decision tree with `max_depth=2`. We would expect this depth to be enough to separate the blobs that are easy to separate.

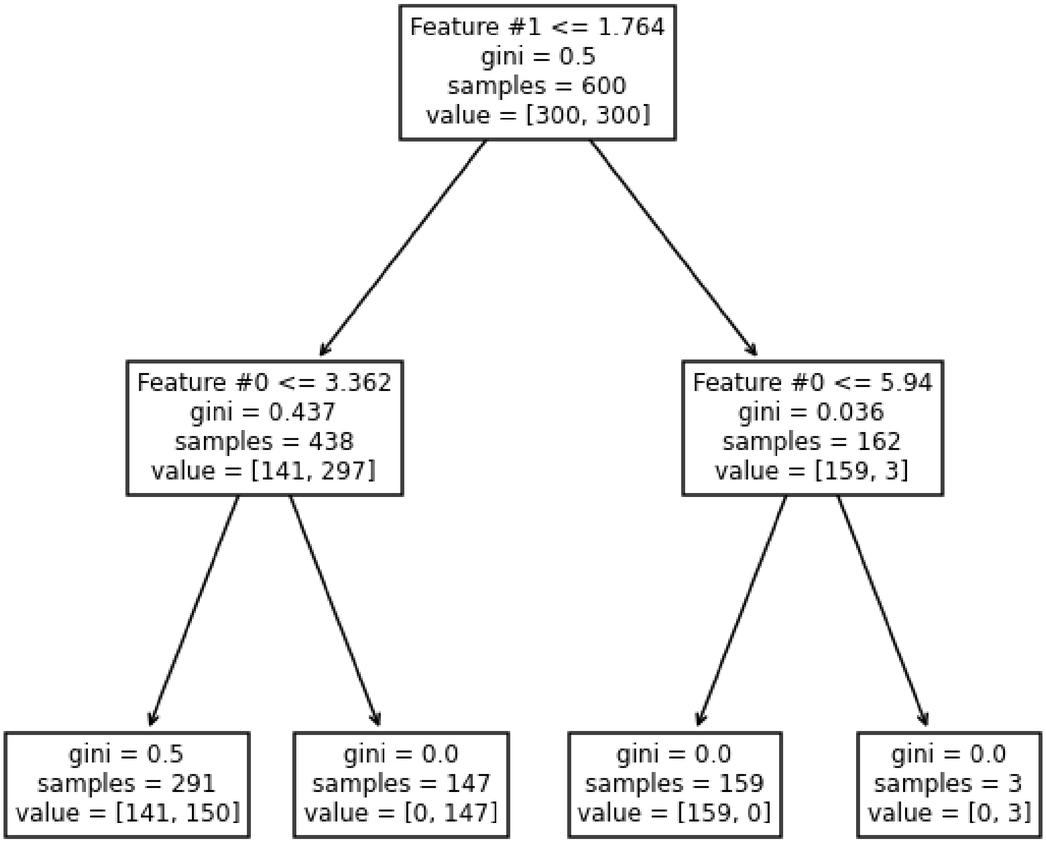
[16]:
```python
max_depth = 2
tree_clf = DecisionTreeClassifier(max_depth=max_depth)
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
_ = plt.title(f"Decision tree with max-depth of {max_depth}")
```



As expected, we see that the blue blob on the right and the red blob on the top are easily separated. However, more splits will be required to better

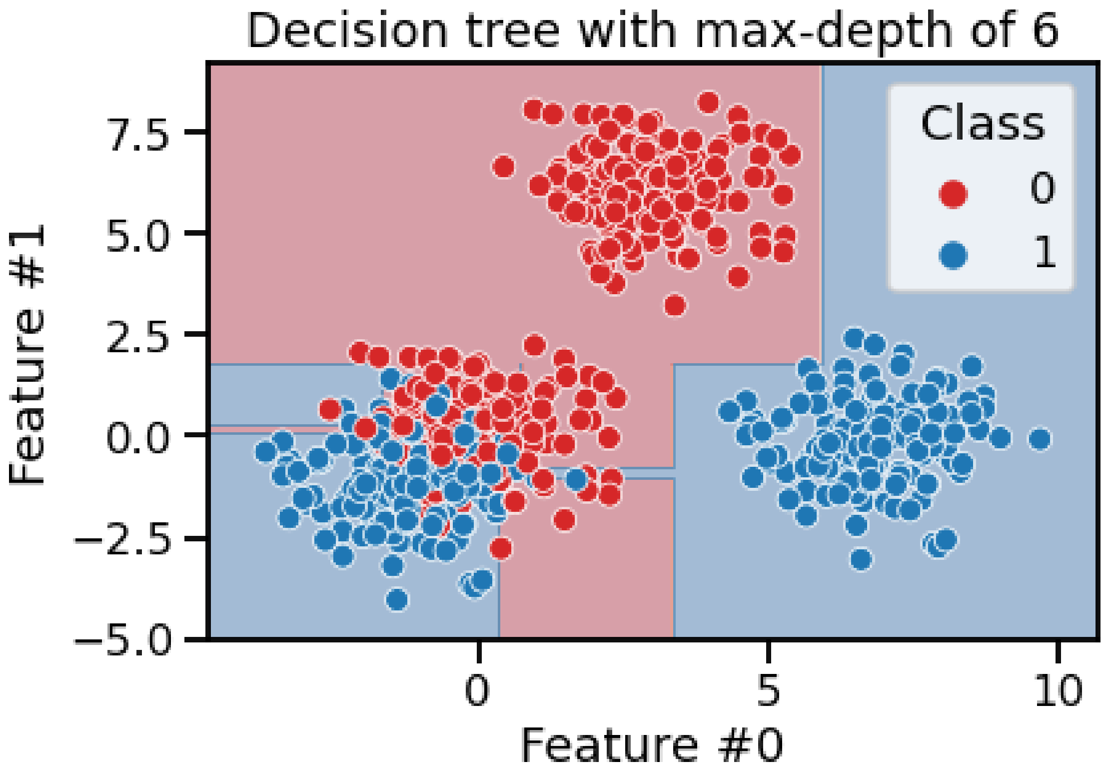
Indeed, we see that red blob on the top and the blue blob on the right of the plot are perfectly separated. However, the tree is still making mistakes in the area where the blobs are mixed together. Let's check the tree representation.

[17]:
```
from sklearn.tree import plot_tree

_, ax = plt.subplots(figsize=(10, 10))
_ = plot_tree(tree_clf, ax=ax, feature_names=data_clf_columns)
```

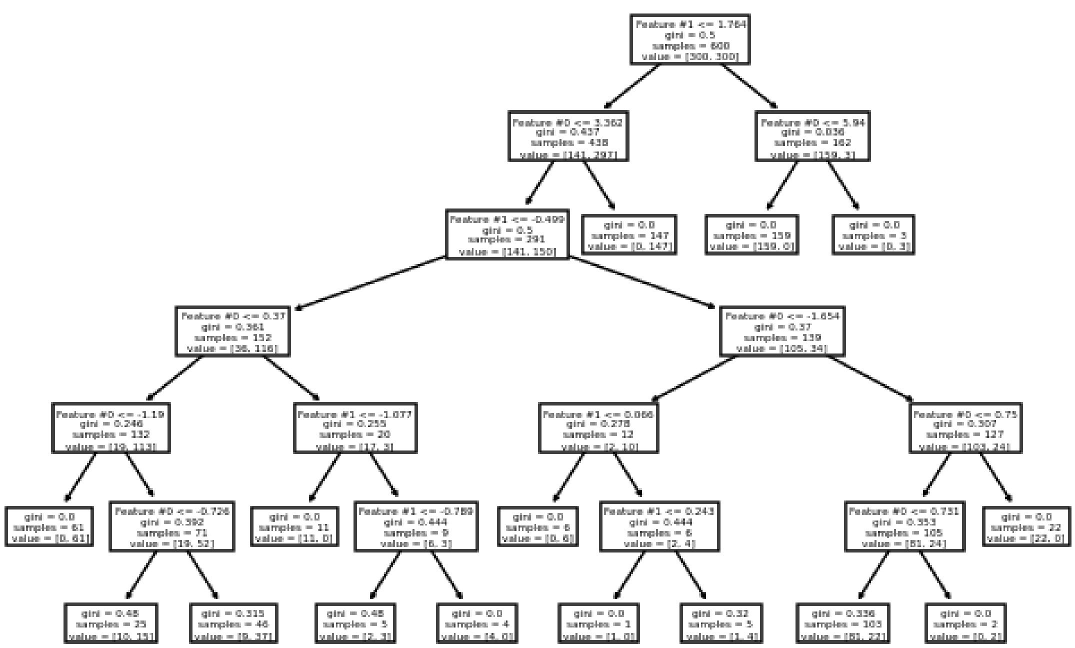


We see that the right branch achieves perfect classification. Now, we increase the depth to check how the tree will grow.

[18]:
```
max_depth = 6
tree_clf = DecisionTreeClassifier(max_depth=max_depth)
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
_ = plt.title(f"Decision tree with max-depth of {max_depth}")
```



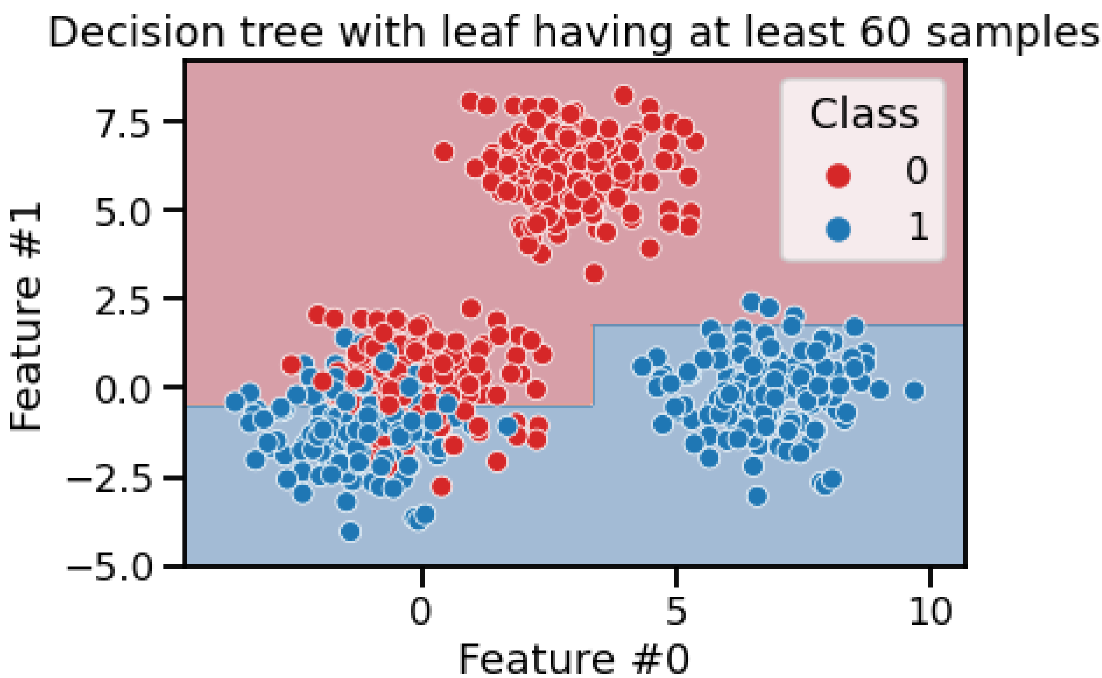
[19]:
```
_, ax = plt.subplots(figsize=(11, 7))
_ = plot_tree(tree_clf, ax=ax, feature_names=data_clf_columns)
```



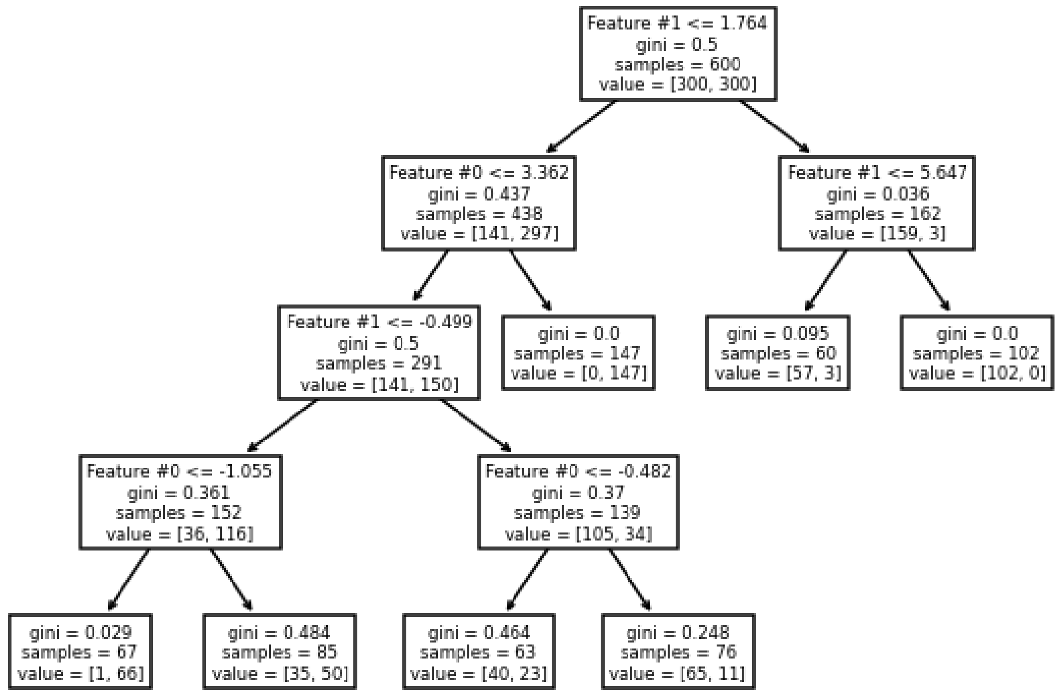
As expected, the left branch of the tree continue to grow while no further splits were done on the right branch. Fixing the `max_depth` parameter would cut the tree horizontally at a specific level, whether or not it would be more beneficial that a branch continue growing.

The hyperparameters `min_samples_leaf`, `min_samples_split`, `max_leaf_nodes`, or `min_impurity_decrease` allows growing asymmetric trees and apply a constraint at the leaves or nodes level. We will check the effect of `min_samples_leaf`.

[20]:
```
min_samples_leaf = 60
tree_clf = DecisionTreeClassifier(min_samples_leaf=min_samples_leaf)
plot_classification(tree_clf, data_clf[data_clf_columns],
                    data_clf[target_clf_column])
_ = plt.title(
    f"Decision tree with leaf having at least {min_samples_leaf} samples")
```



[21]:
```
_, ax = plt.subplots(figsize=(10, 7))
_ = plot_tree(tree_clf, ax=ax, feature_names=data_clf_columns)
```



This hyperparameter allows to have leaves with a minimum number of samples and no further splits will be search otherwise. Therefore, these hyperparameters could be an alternative to fix the `max_depth` hyperparameter.