In [1]:
```
import mglearn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Naive Bayes Classifiers

Naive Bayes models often provide generalization performance that is slightly worse than that of linear classifiers like LogisticRegression and LinearSVC.

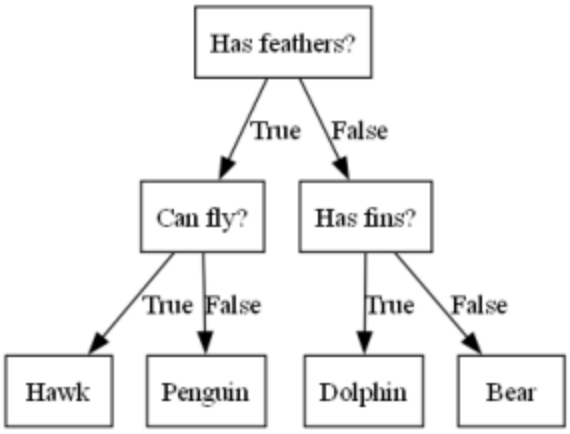
There are three kinds of naive Bayes classifiers implemented in scikit-learn:

1. GaussianNB : continuous data
2. BernoulliNB : binary data
3. MultinomialNB : count data

Decision Trees

Decision trees are widely used models for classification and regression tasks.

In [2]:
```
mglearn.plots.plot_animal_tree()
```



There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called pre-pruning), or building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning).

In [3]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

In [4]:
```
X_train, X_test, y_train, y_test = train_test_split(
 cancer.data, cancer.target, stratify=cancer.target, random_state=42)
```

In [5]:
```
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
```

Out[5]:
```
DecisionTreeClassifier(random_state=0)
```

In [6]:
```
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data.

In [7]:
```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)
```

Out[7]:
```
DecisionTreeClassifier(max_depth=4, random_state=0)
```

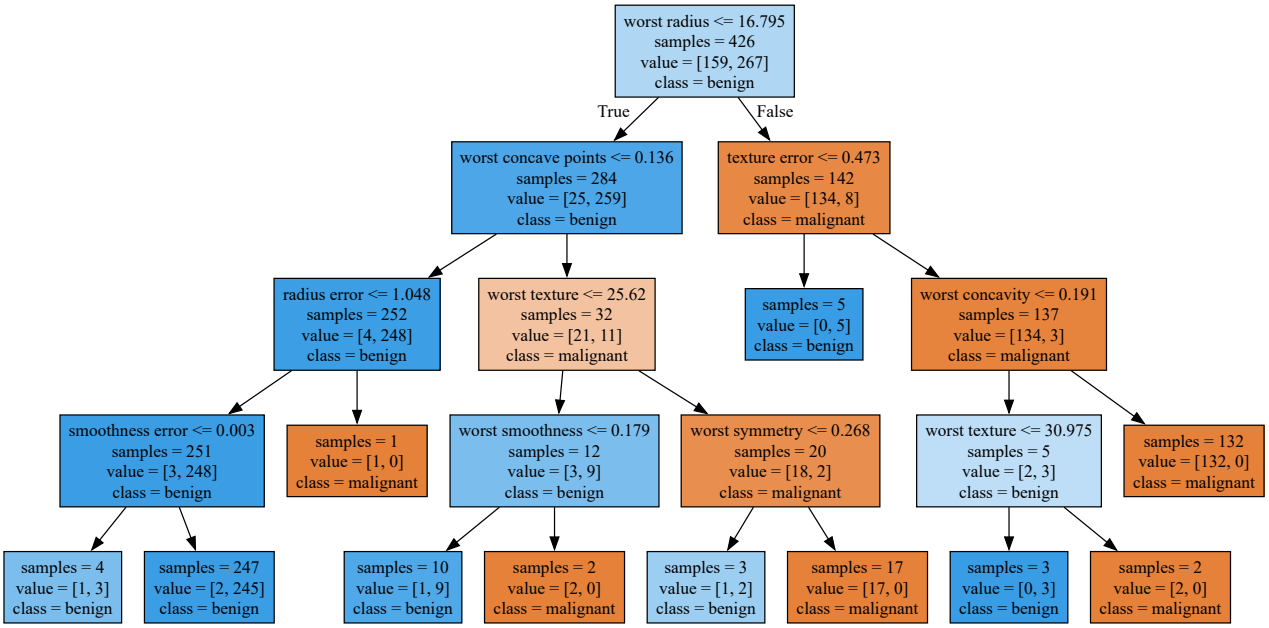
In [8]:
```
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set.

In [9]:
```
import graphviz
with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Out[9]:



In [10]:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

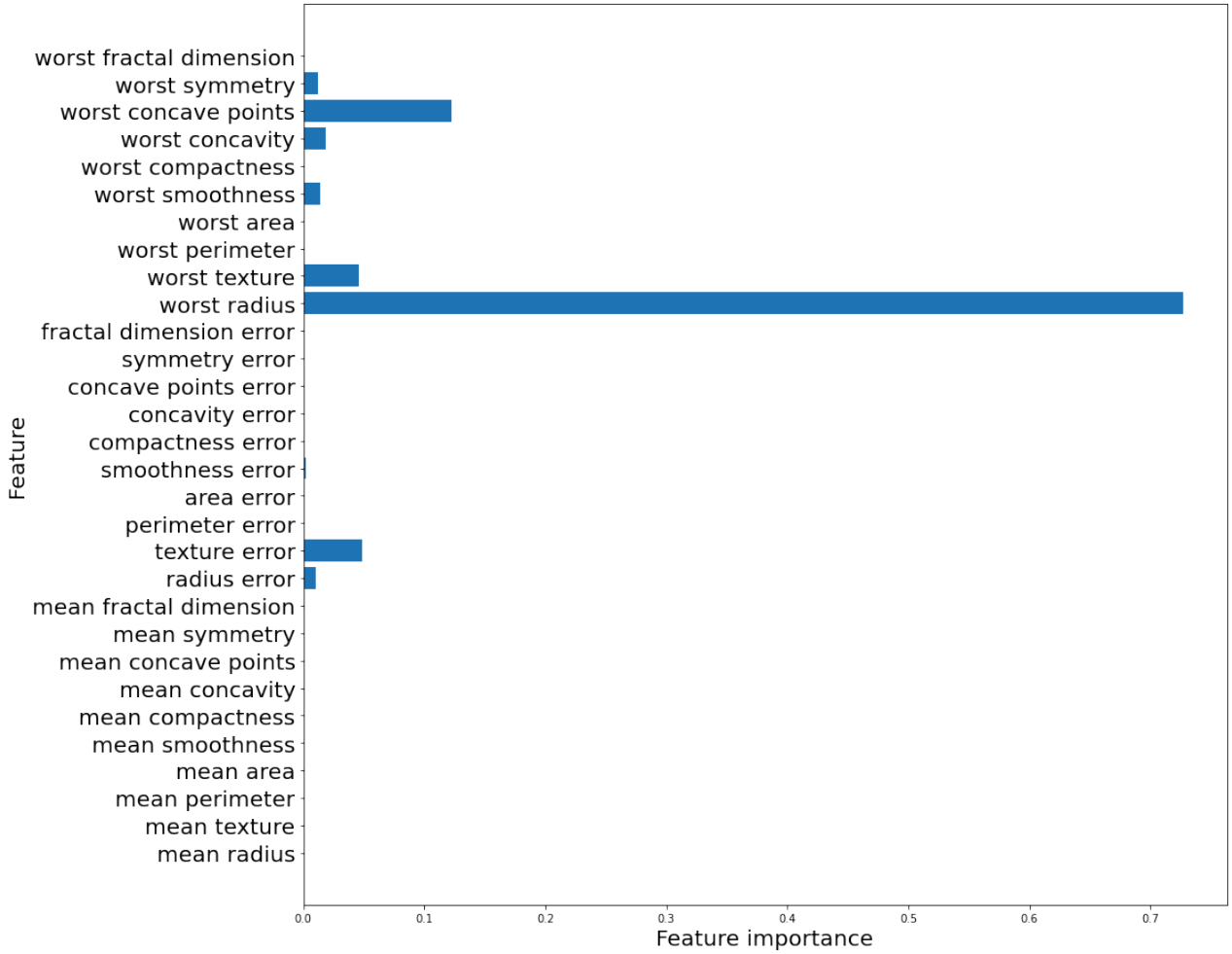
```
Feature importances:
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.01019737 0.04839825
 0.         0.         0.0024156  0.         0.         0.
 0.         0.         0.72682851 0.0458159  0.         0.
 0.0141577  0.         0.018188   0.1221132  0.01188548 0.        ]
```

Instead of looking at the whole tree, which can be taxing, there are some useful prop- erties that we can derive to summarize the workings of the tree. The most commonly used summary is feature importance, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target.” The feature importances always sum to 1.

In [11]:

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.figure(figsize=(15,15))
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names, fontsize=20)
    plt.xlabel("Feature importance", fontsize=20)
    plt.ylabel("Feature", fontsize=20 )

plot_feature_importances_cancer(tree)
```



Here we see that the feature used in the top split ("worst radius") is by far the most important feature. This confirms our observation in analyzing the tree that the first level already separates the two classes fairly well.

However, if a feature has a low feature_importance, it doesn't mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.

Usually, picking one of the pre-pruning strategies—setting either max_depth, max_leaf_nodes, or min_samples_leaf—is sufficient to prevent overfitting.

Random Forest

The number of features that are selected is controlled by the max_features parameter.

A high max_features means that the trees in the random forest will be quite similar, and they will be able to fit the data easily, using the most distinctive features.

A low max_features means that the trees in the random forest will be quite different, and that each tree might need to be very deep in order to fit the data well.

In [12]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons
```

In [13]:
```
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
```

In [14]:
```
forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

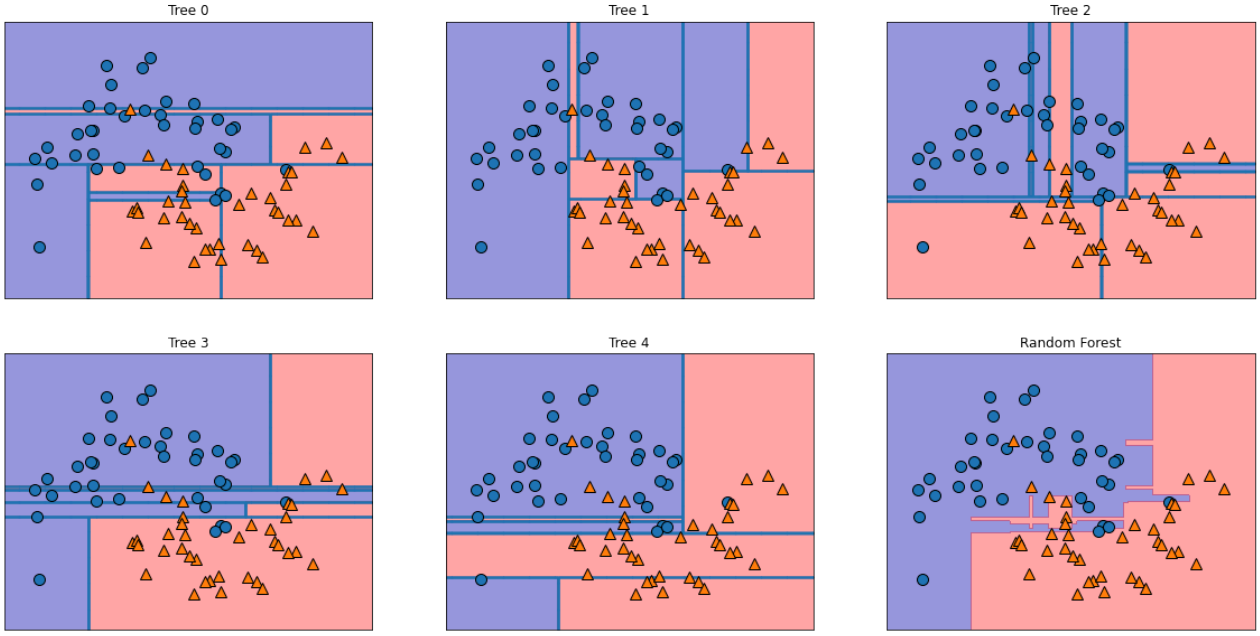
Out[14]:
```
RandomForestClassifier(n_estimators=5, random_state=2)
```

In [15]:
```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest,
                                X_train,
                                fill=True,
                                ax=axes[-1, -1],
                                alpha=.4)

axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

Out[15]:
```
[<matplotlib.lines.Line2D at 0x1d7339129d0>,
 <matplotlib.lines.Line2D at 0x1d7338c0ca0>]
```



We would use many more trees (often hundreds or thousands), leading to even smoother boundaries.

In [16]:
```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_
```

```
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)
```

Out[16]:
```
RandomForestClassifier(random_state=0)
```

In [17]:
```
print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

Gradient Boosted Regression Trees

They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly.

Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the learning_rate, which controls how strongly each tree tries to correct the mistakes of the previous trees.

A higher learning rate means each tree can make stronger corrections, allowing for more complex models.

In [18]:
```
from sklearn.ensemble import GradientBoostingClassifier
```

In [19]:
```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
```

In [20]:
```
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)
```

Out[20]:
```
GradientBoostingClassifier(random_state=0)
```

In [21]:
```
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.965
```

As the training set accuracy is 100%, we are likely to be overfitting. To reduce overfitting, we could either apply stronger pre-pruning by limiting the maximum depth or lower the learning rate.

In [22]:
```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)
```

Out[22]:
```
GradientBoostingClassifier(max_depth=1, random_state=0)
```

In [23]:
```
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

In [24]:
```
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)
```

Out[24]:
```
GradientBoostingClassifier(learning_rate=0.01, random_state=0)
```

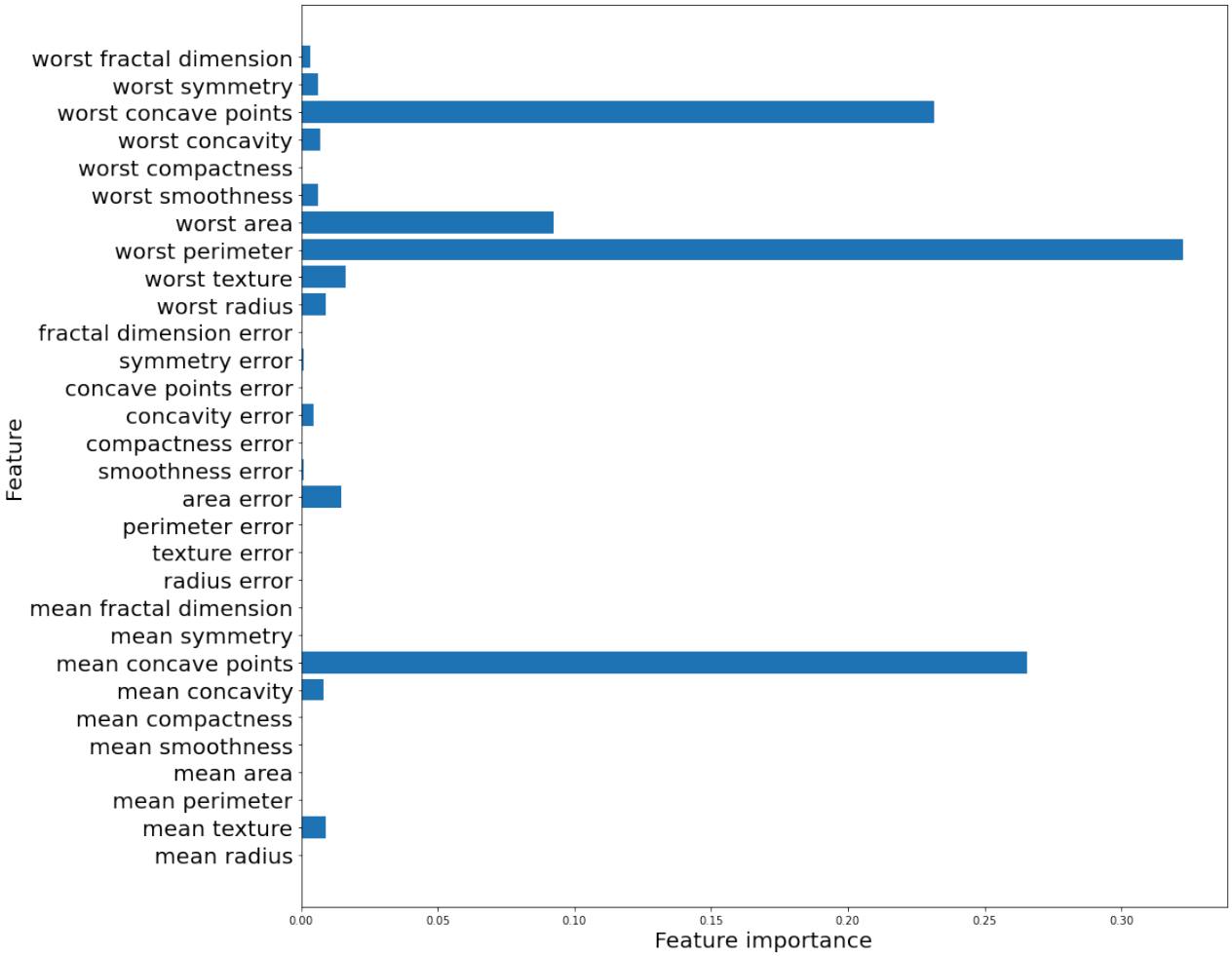
In [25]:
```
print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Both methods of decreasing the model complexity reduced the training set accuracy, as expected. In this case, lowering the maximum depth of the trees provided a signifi- cant improvement of the model, while lowering the learning rate only increased the generalization performance slightly.

In [26]:
```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

plot_feature_importances_cancer(gbrt)
```



As both gradient boosting and random forests perform well on similar kinds of data, a common approach is to first try random forests, which work quite robustly. If random forests work well but prediction time is at a premium, or it is important to squeeze out the last percentage of accuracy from the machine learning model, moving to gradient boosting often helps.