Exercise 1

在一个有100万个实例的训练集上，训练了（无限制）的决策树的近似深度是多少？

答案：

包含 m 个叶子节点的均衡二叉树的深度等于 $log_2(m)$ ，向上取整。log_2 是以2为底的对数；$log_2(m) = log(m)/log(2)$ 。一棵二叉决策树（只做二元决策的树，就像 Scikit-Learn 中的所有树一样）在训练结束时或多或少会达到平衡，如果不受限制地训练，每个训练实例有一个叶子。因此，如果训练集包含一百万个实例，则决策树的深度将为 $log_2(10^6) \approx 20$（实际上多一点，因为树通常不会完全平衡）。

Exercise 2

一个节点的基尼不纯度通常是低于还是高于它的父节点？它通常是低/高，还是总是低/高？

答案：

节点的基尼不纯度通常低于其父节点。这是由于 CART 训练算法的成本函数，它以最小化其子节点基尼不纯度的加权和的方式拆分每个节点。但是，一个节点有可能比其父节点具有更高的 Gini 不纯度，只要这种增加被另一个子节点杂质的减少所补偿。

例如，考虑包含四个 A 类实例和一个 B 类实例的节点。其基尼不纯度为 $1–(1/5)^2–(4/5)^2 = 0.32$ 。现在假设数据集是一维的，实例按以下顺序排列：A、B、A、A、A。您可以验证该算法将在第二个实例之后拆分此节点，生成一个包含实例的子节点A、B 和另一个具有实例 A、A、A 的子节点。第一个子节点的基尼不纯度为 $1–(1/2)^2–(1/2)^2 = 0.5$ ，高于其父节点。另一个节点是纯节点这一事实对此进行了补偿，因此其总体加权基尼不纯度为 $2/5 \times 0.5 + 3/5 \times 0 = 0.2$ ，低于父节点的基尼不纯度。

Exercise 3

如果决策树过拟合训练集，尝试减少 **max_depth** 是一个好主意吗？

答案：

如果决策树过度拟合训练集，减少 max_depth 可能是个好主意，因为这会限制模型，使其正则化。

Exercise 4

如果一个决策树欠拟合训练集，那么尝试缩放输入特征是一个好主意吗？

答案：

决策树不关心训练数据是否缩放或居中；这是他们的优点之一。因此，如果决策树欠适合训练集，则缩放输入特征只会浪费时间。

Exercise 5

如果在一个包含100万个实例的训练集上训练一个决策树需要一个小时，那么在一个包含1000万个实例的训练

集上训练另一个决策树大约需要多少时间呢？提示：考虑 CART 算法的计算复杂度。

答案：

训练决策树的计算复杂度为 $O(n \times m\ log_2(m))$ 。因此，如果将训练集大小乘以10，则训练时间将乘以 $K = (n \times 10\ m \times log2(10\ m))/(n \times m \times log_2(m)) = 10 \times log_2(10\ m)/log_2(m)$ 。如果 $m = 10^6$ ，则 $K \approx 11.7$ ，因此您可以预计训练时间约为 11.7 小时。

Exercise 6

如果在一个给定的训练集上训练一个决策树需要一个小时，那么如果你将特性的数量增加一倍，大约需要多少时间呢？

答案：

如果特征数量翻倍，那么训练时间也会大致翻倍。

Exercise 7

通过以下步骤来训练和微调卫星数据集的决策树：

1. 使用 **make_moons(n_samples=10000, noise=0.4)** 来生成一个卫星数据集。
2. 使用 **train_test_split()** 将数据集分割为训练集和测试集。
3. 使用带有交叉验证的网格搜索（借助 **GridSearchCV** 类的帮助）来为 **DecisionTreeClassifier** 找到良好的超参数值。提示：尝试使用 **max_leaf_nodes** 的各种值。
4. 使用这些超参数在完整的训练集上进行训练，并测量模型在测试集上的性能。你应该得到大约85%到87%的准确率。

答案：

添加 `random_state=42` 以使该笔记本的输出保持不变：

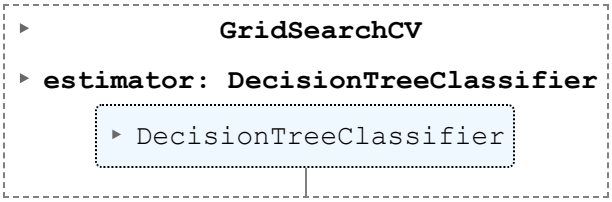
In [1]:
```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=10000, noise=0.4, random_state=42)
```

In [2]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_moons, y_moons,
                                                    test_size=0.2,
                                                    random_state=42)
```

In [4]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

params = {
    'max_leaf_nodes': list(range(2, 100)),
    'max_depth': list(range(1, 7)),
    'min_samples_split': [2, 3, 4]
}
grid_search_cv = GridSearchCV(DecisionTreeClassifier(random_state=42),
                              params,
                              cv=3)
```

```
grid_search_cv.fit(X_train, y_train)
```

Out[4]:



In [5]:
```
grid_search_cv.best_estimator_
```

Out[5]:
```
DecisionTreeClassifier(max_depth=6, max_leaf_nodes=17, random_state=42)
```

默认情况下，**GridSearchCV** 会训练在整个训练集上找到的最佳模型（您可以通过设置 **refit=False** 来更改此设置），因此我们无需再次执行此操作。我们可以简单地评估模型的准确性：

In [6]:
```
from sklearn.metrics import accuracy_score

y_pred = grid_search_cv.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[6]:
```
0.8595
```

Exercise 8

通过执行以下步骤来形成一个森林：

1. 继续前面的练习，生成训练集的1000个子集，每个子集包含随机选择的100个实例。提示：你可以使用ScikitLearn的 **ShuffleSplit** 类。
2. 使用上一个练习中找到的最佳超参数值，在每个子集上训练一个决策树。在测试集上评估这1000棵决策树。由于它们是在较小的集合上训练的，这些决策树可能比第一个决策树表现得更差，仅达到大约80%的准确率。
3. 现在魔术来了。对于每个测试集实例，生成1000棵决策树的预测，并只保留最频繁的预测（您可以为此使用SciPy的 **mode()** 函数）。这种方法为您对测试集的 **多数投票预测（majority-vote predictions）** 。
4. 在测试集上评估这些预测：您应该获得比第一个模型略高一些的精度（大约高出0.5到1.5%）。恭喜你，你已经训练好了一个随机的森林分类器！

In [8]:
```
from sklearn.model_selection import ShuffleSplit

n_trees = 1000
n_instances = 100

mini_sets = []

rs = ShuffleSplit(n_splits=n_trees, test_size=len(X_train) - n_instances,
                  random_state=42)

for mini_train_index, mini_test_index in rs.split(X_train):
    X_mini_train = X_train[mini_train_index]
    y_mini_train = y_train[mini_train_index]
    mini_sets.append((X_mini_train, y_mini_train))
```

In [16]:
```
from sklearn.base import clone
import numpy as np

forest = [clone(grid_search_cv.best_estimator_) for _ in range(n_trees)]

accuracy_scores = []

for tree, (X_mini_train, y_mini_train) in zip(forest, mini_sets):
    tree.fit(X_mini_train, y_mini_train)

    y_pred = tree.predict(X_test)
    accuracy_scores.append(accuracy_score(y_test, y_pred))

np.mean(accuracy_scores)
```

Out[16]:
```
0.8056605
```

In [17]:
```
Y_pred = np.empty([n_trees, len(X_test)], dtype=np.uint8)

for tree_index, tree in enumerate(forest):
    Y_pred[tree_index] = tree.predict(X_test)
```

In [31]:
```
from scipy.stats import mode

y_pred_majority_votes, n_votes = mode(Y_pred, axis=0, keepdims=True)
```

In [32]:
```
accuracy_score(y_test, y_pred_majority_votes.reshape([-1]))
```

Out[32]:
```
0.873
```