# 决策树和集成方法

- trees and forests

# 决策树

- 决策树是一种多功能机器学习算法， 即可以执行分类任务也可以执行回归任务， 它是一种功能很强大的算法，可以对很复杂的数据集进行拟合。
- 在本节我们将首先讨论如何使用决策树进行训练，可视化和预测。然后我们会学习在 Scikit-learn 上面使用 CART 算法，并且探讨如何调整决策树让它可以用于执行回归任务。最后，我们当然也需要讨论一下决策树目前存在的一些局限性。

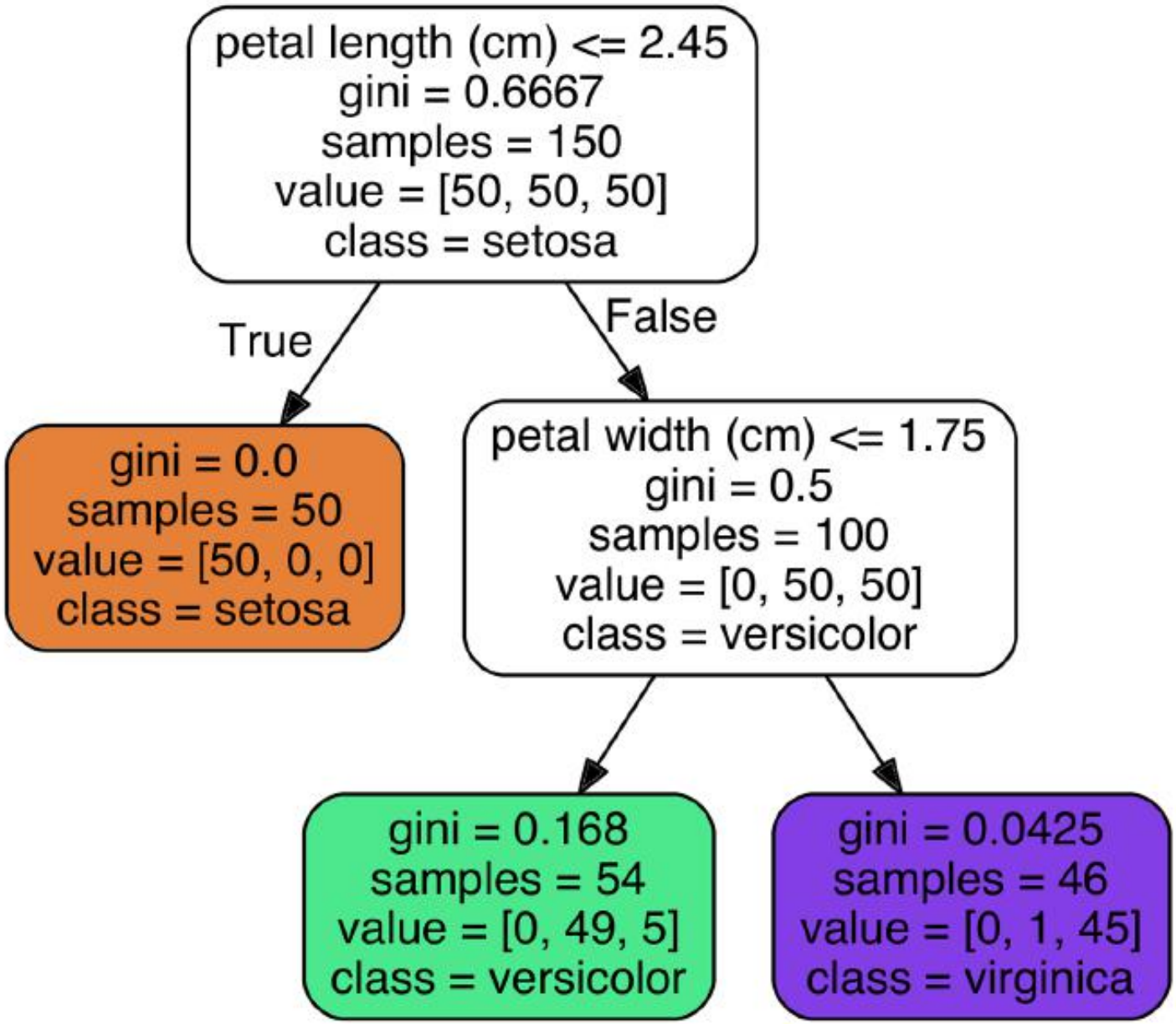
# 决策树的训练和可视化

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

# 决策树的训练和可视化

```
from sklearn.tree import export_graphviz
export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

# $ dot -Tpng iris_tree.dot -o iris_tree.png



# 开始预测

- 假设你找到了一朵鸢尾花并且想对它进行分类，你从根节点开始（深度为 0，顶部）：该节点询问花朵的花瓣长度是否小于 2.45 厘米。如果是，您将向下移动到根的左侧子节点（深度为 1，左侧）。 在这种情况下，它是一片叶子节点（即它没有任何子节点），所以它不会问任何问题：你可以方便地查看该节点的预测类别，决策树预测你的花是 Iris-Setosa（class = setosa）。

# 开始预测

- 节点的samples属性统计出它应用于多少个训练样本实例。节点的value属性告诉你这个节点对于每一个类别的样例有多少个。最后，节点的Gini属性用于测量它的纯度：如果一个节点包含的所有训练样例全都是同一类别的，我们就说这个节点是纯的（Gini=0）。

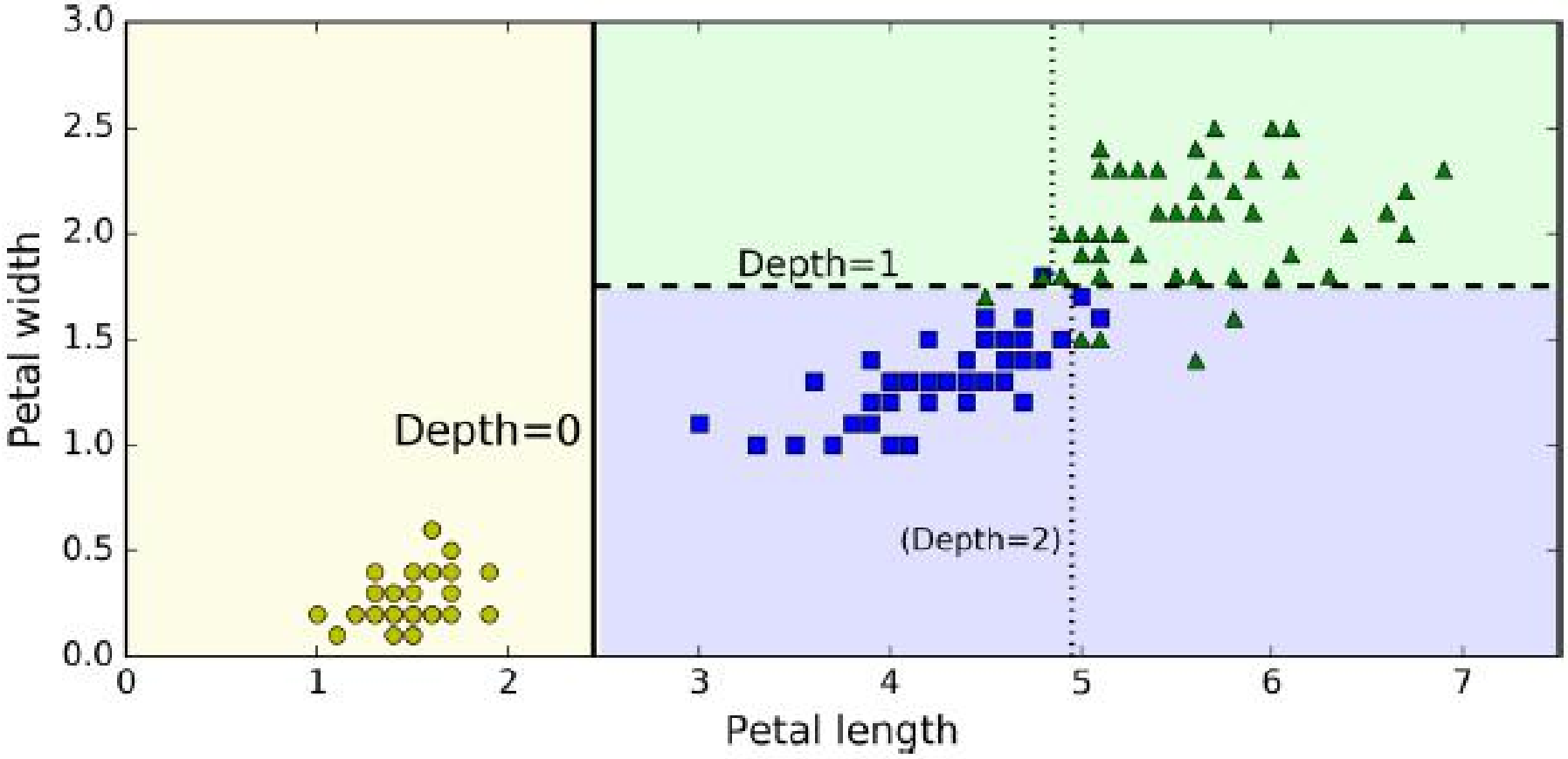
# 开始预测

- 公式 6-1 显示了训练算法如何计算第i个节点的 gini 分数 G_i。例如， 深度为 2 的左侧节点基尼指数为：$1 – (0/54)^2 – (49/54)^2 – (5/54)^2 \approx 0.168$.

*Equation 6-1. Gini impurity*

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

# 决策树的决策边界


Depth=1
Depth=0
(Depth=2)
Petal width
Petal length
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
1
2
3
4
5
6
7

# 估计分类概率

- 决策树还可以估计某个实例属于特定类k的概率：首先遍历树来查找此实例的叶节点，然后它返回此节点中类k的训练实例的比例。

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. , 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

# CART 训练算法

- Scikit-Learn 用分类回归树（Classification And Regression Tree，简称 CART）算法训练决策树。这种算法思想真的非常简单：首先使用单个特征k和阈值 t_k（例如，“花瓣长度≤2.45cm”）将训练集分成两个子集。它如何选择k和 t_k 呢？它寻找到能够产生最纯粹的子集一对(k, t_k)，然后通过子集大小加权计算。

*Equation 6-2. CART cost function for classification*

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

# CART 训练算法

- 当它成功的将训练集分成两部分之后， 它将会继续使用相同的递归式逻辑继续的分割子集，然后是子集的子集。当达到预定的最大深度之后将会停止分裂（由max_depth超参数决定），或者是它找不到可以继续降低不纯度的分裂方法的时候。
- 不幸的是，找到最优树是一个 NP 完全问题：需要 $O(exp^m)$ 时间，即使对于相当小的训练集也会使问题变得棘手。 这就是为什么我们必须设置一个“合理的”（而不是最佳的）解决方案。

# 计算复杂度

- 在建立好决策树模型后， 做出预测需要遍历决策树， 从根节点一直到叶节点。决策树通常近似左右平衡，因此遍历决策树需要经历大致$O(log_2(m))$个节点。由于每个节点只需要检查一个特征的值，因此总体预测复杂度仅为$O(log_2(m))$，与特征的数量无关。 所以即使在处理大型训练集时，预测速度也非常快。

# 计算复杂度

- 然而，训练算法的时候（训练和预测不同）需要在每个节点的所有样本上比较所有特征（如果设置了max_features会更少一些）。就有了 O(n×m log(m)) 的训练复杂度。对于小型训练集（少于几千例），Scikit-Learn 可以通过预先设置数据（presort = True）来加速训练，但是这对于较大训练集来说会显着减慢训练速度。

# 基尼不纯度还是信息熵?

- 熵的概念是源于热力学中分子混乱程度的概念，当分子井然有序的时候，熵值接近于 0。熵这个概念后来逐渐被扩展到了各个领域，其中包括香农的信息理论，这个理论被用于测算一段信息中的平均信息密度。当所有信息相同的时候熵被定义为零。在机器学习中，熵经常被用作不纯度的衡量方式，当一个集合内只包含一类实例时， 我们称为数据集的熵为 0。熵的减少通常称为信息增益。

*Equation 6-3. Entropy*

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log\left(p_{i,k}\right)$$

# 基尼不纯度还是信息熵?

- 那么我们到底应该使用 Gini 指数还是熵呢？ 事实上大部分情况都没有多大的差别：他们会生成类似的决策树。基尼指数计算稍微快一点，所以这是一个很好的默认值。但是，也有的时候它们会产生不同的树，基尼指数会趋于在树的分支中将最多的类隔离出来，而熵指数趋向于产生略微平衡一些的决策树模型。

*Equation 6-1. Gini impurity*

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

*Equation 6-3. Entropy*

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log\left(p_{i,k}\right)$$

# 正则化超参数

- 决策树几乎不对训练数据做任何假设（于此相反的是线性回归等模型，这类模型通常会假设数据是符合线性关系的）。如果不添加约束，树结构模型通常将根据训练数据调整自己，使自身能够很好的拟合数据，而这种情况下大多数会导致模型过拟合。
- 这一类的模型通常会被称为非参数模型，这不是因为它没有任何参数（通常也有很多），而是因为在训练之前没有确定参数的具体数量，所以模型结构可以根据数据的特性自由生长。
- 于此相反的是，像线性回归这样的参数模型有事先设定好的参数数量，所以自由度是受限的，这就减少了过拟合的风险（但是增加了欠拟合的风险）。

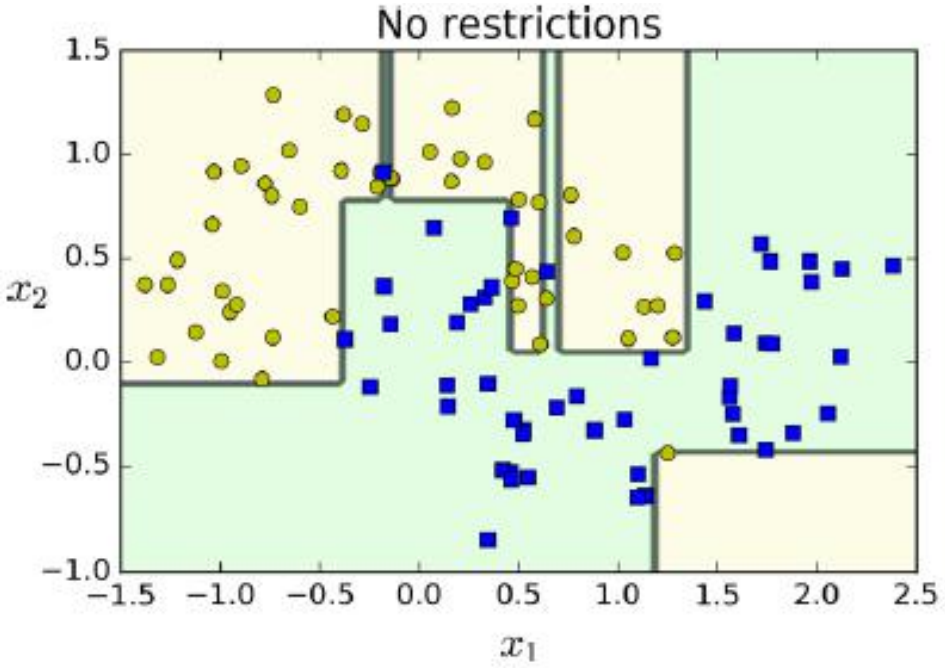
# 正则化超参数

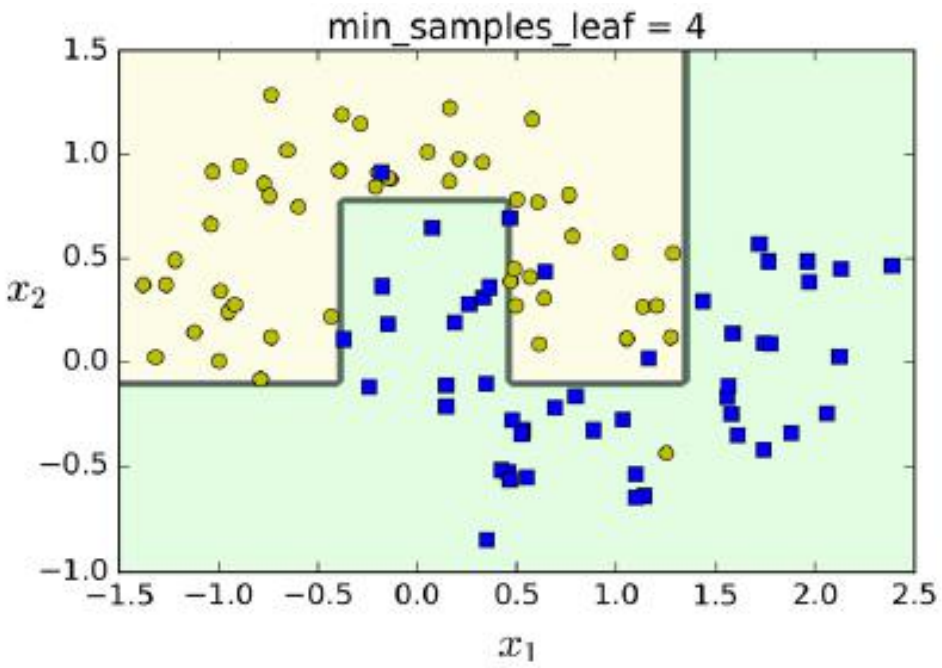
- 为了避免过拟合训练数据，我们需要在训练时限制决策树的自由度，即使用正则化技术。最常用的正则化超参数涉及限制决策树的深度。在Scikit-Learn里面通过max_depth 超参数控制(缺省值为 None, 表示无限制)。降低max_depth可以对模型正则化并降低过拟合的风险。

# 正则化超参数

- DecisionTreeClassifier 类有一些其他参数可以限制决策树的形状: min_samples_split（节点在被分裂之前必须具有的最小样本数），min_samples_leaf（叶节点必须具有的最小样本数），max_leaf_nodes（叶节点的最大数量）和max_features（在每个节点被评估是否分裂的时候，具有的最大特征数量）。增加min_* 超参数或者减少max_* 超参数会使模型正则化。

# 使用 min_samples_leaf 正则化



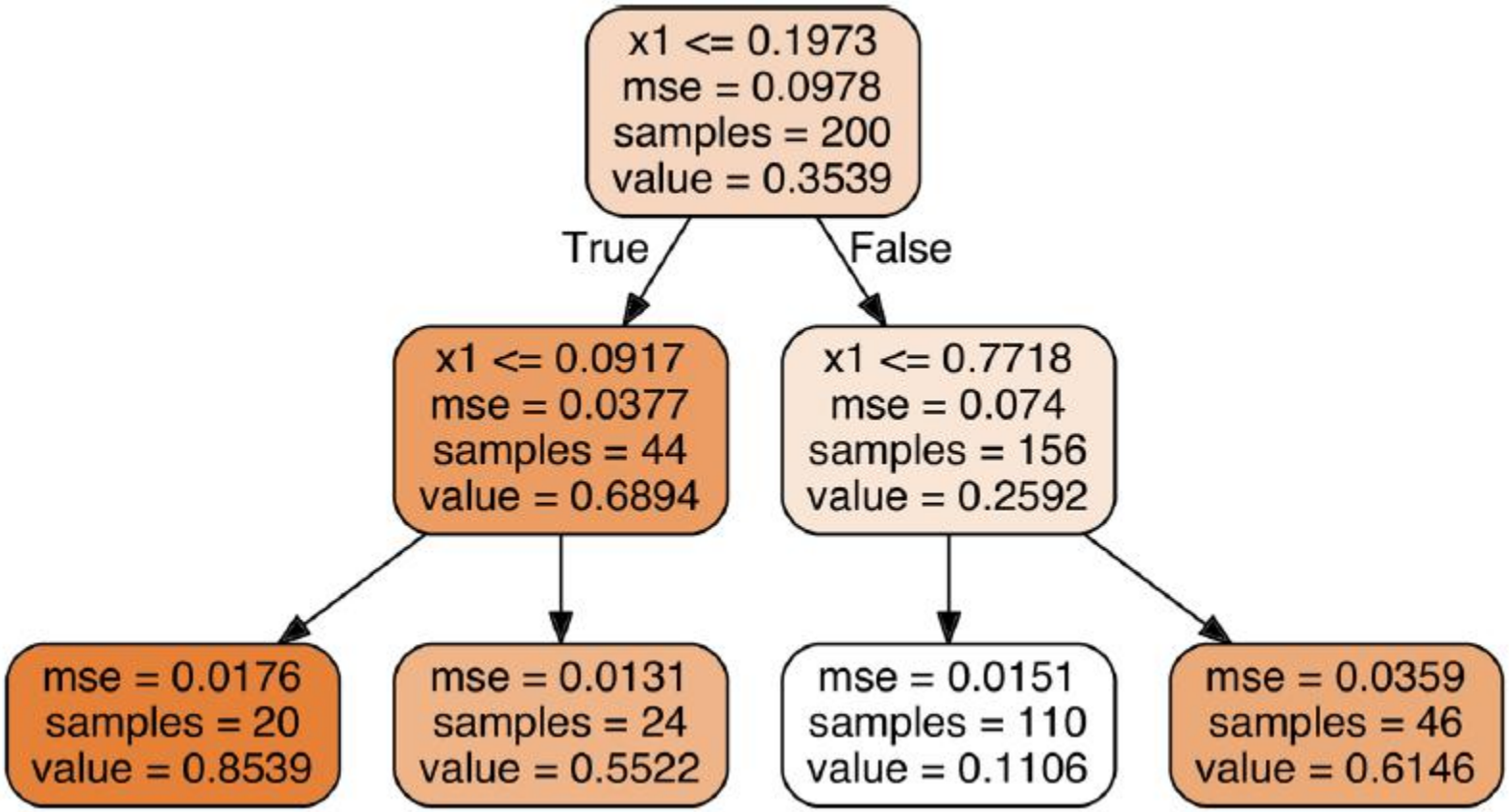



# 回归

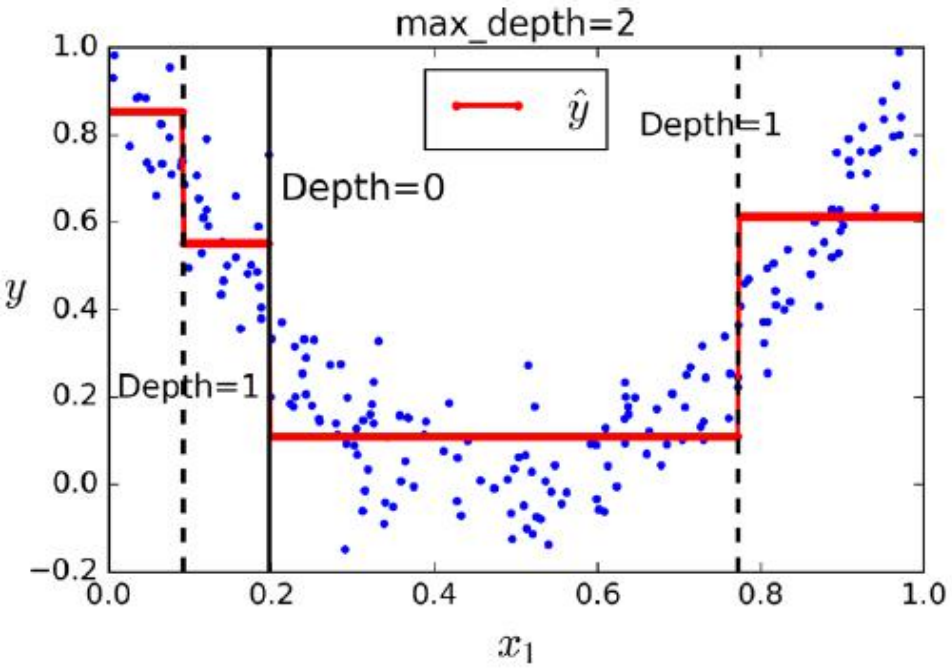
**from sklearn.tree import** DecisionTreeRegressor

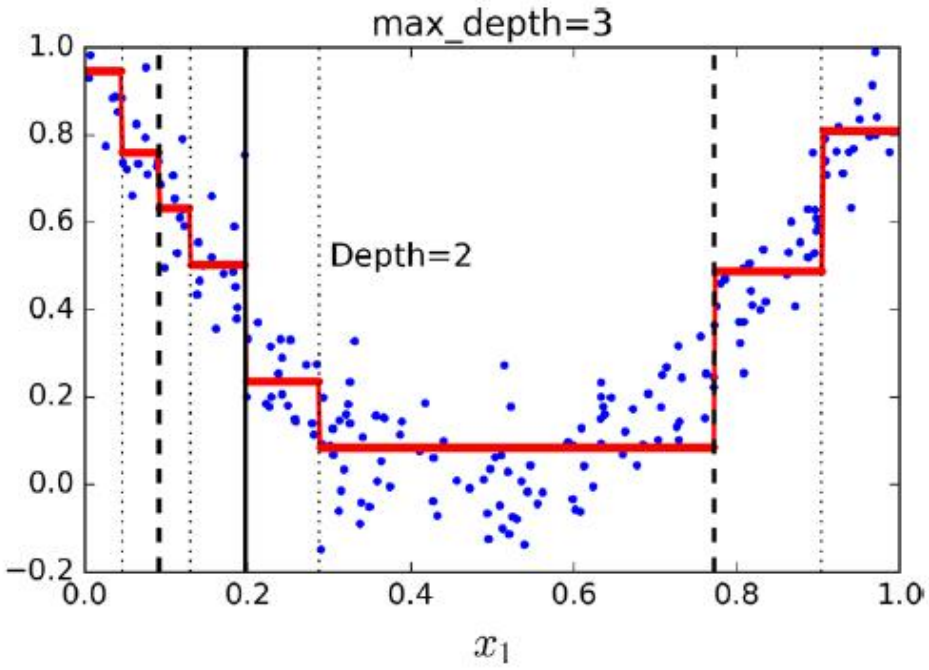
tree_reg = DecisionTreeRegressor(max_depth=2)

tree_reg.fit(X, y)



# 两个决策树回归模型的预测对比



# 回归

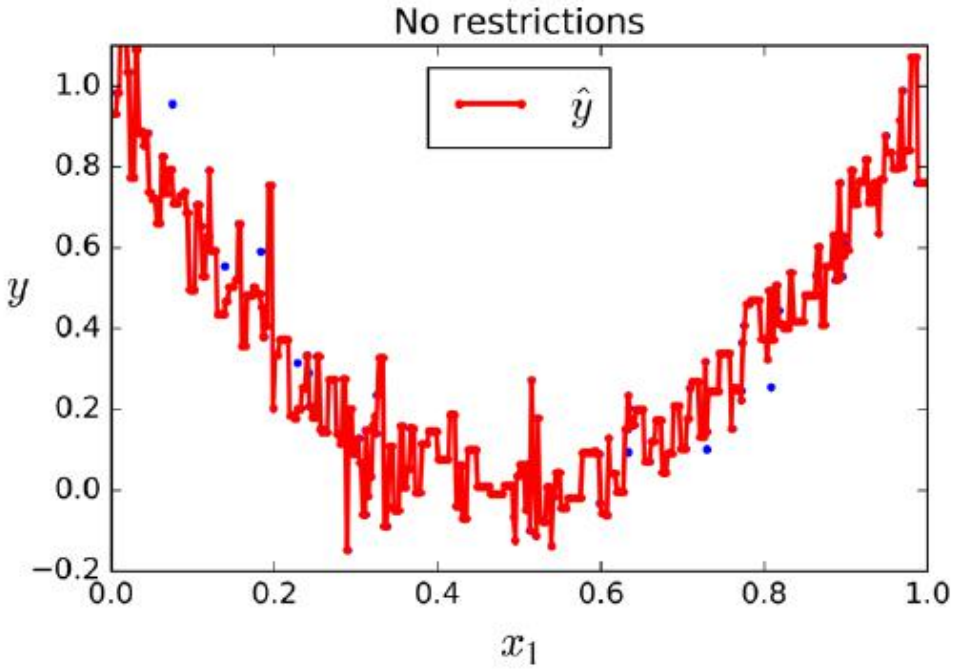
- CART 算法的工作方式与之前处理分类模型基本一样，不同之处在于，现在不再以最小化不纯度的方式分割训练集，而是试图以最小化 MSE 的方式分割训练集。公式 6-4 显示了成本函数，该算法试图最小化这个成本函数。

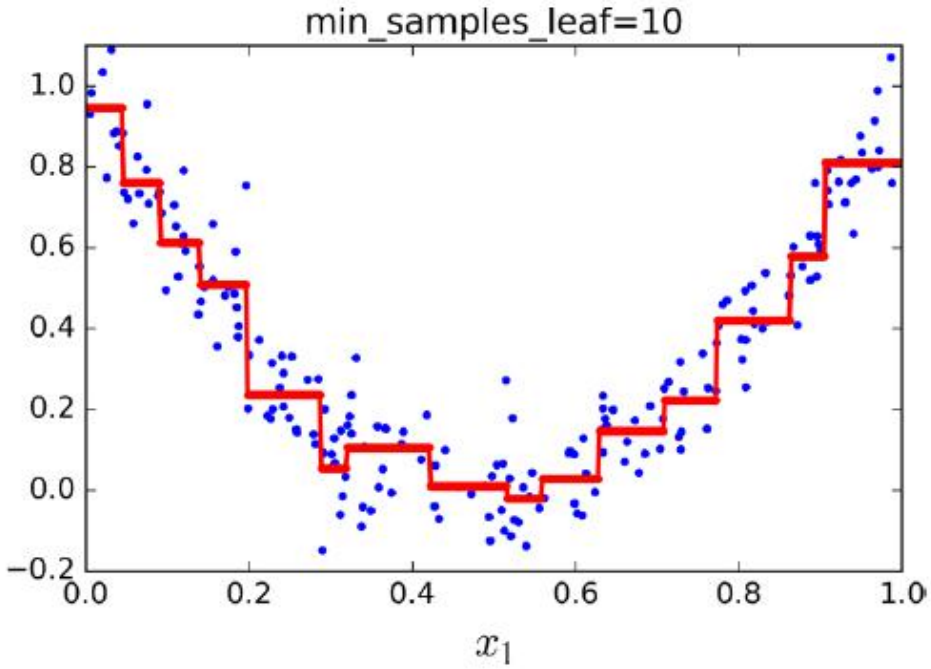
*Equation 6-4. CART cost function for regression*

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

# 回归

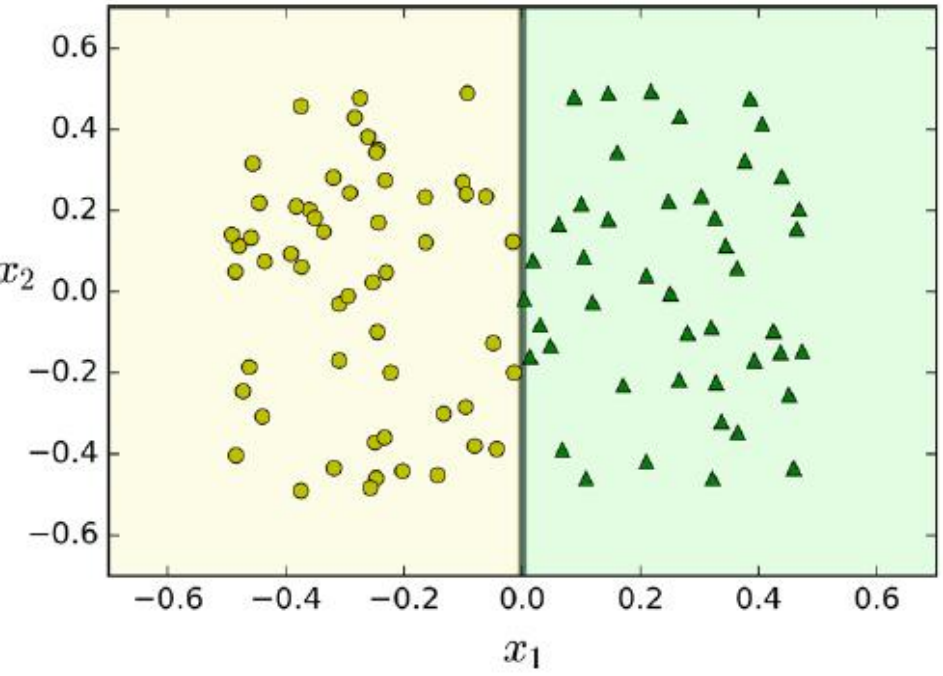
- 和处理分类任务时一样，决策树在处理回归问题的时候也容易过拟合。如果不添加任何正则化（默认的超参数），你就会得到图 6-6 左侧的预测结果，显然，过度拟合的程度非常严重。而当我们设置了 min_samples_leaf = 10，相对就会产生一个更加合适的模型了，就如右图所示的那样。

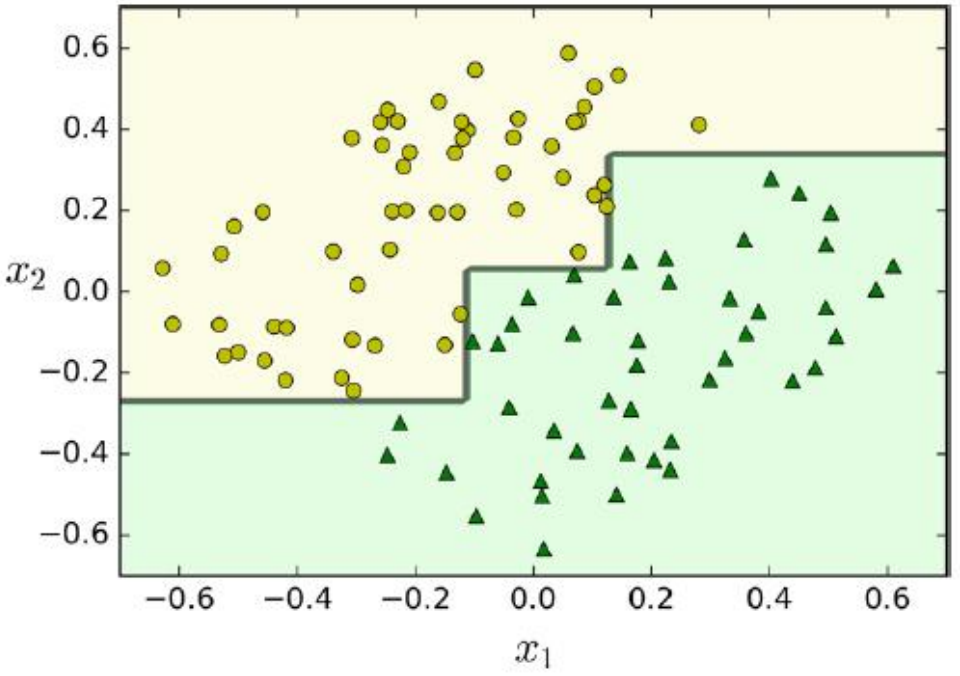


# 不稳定性

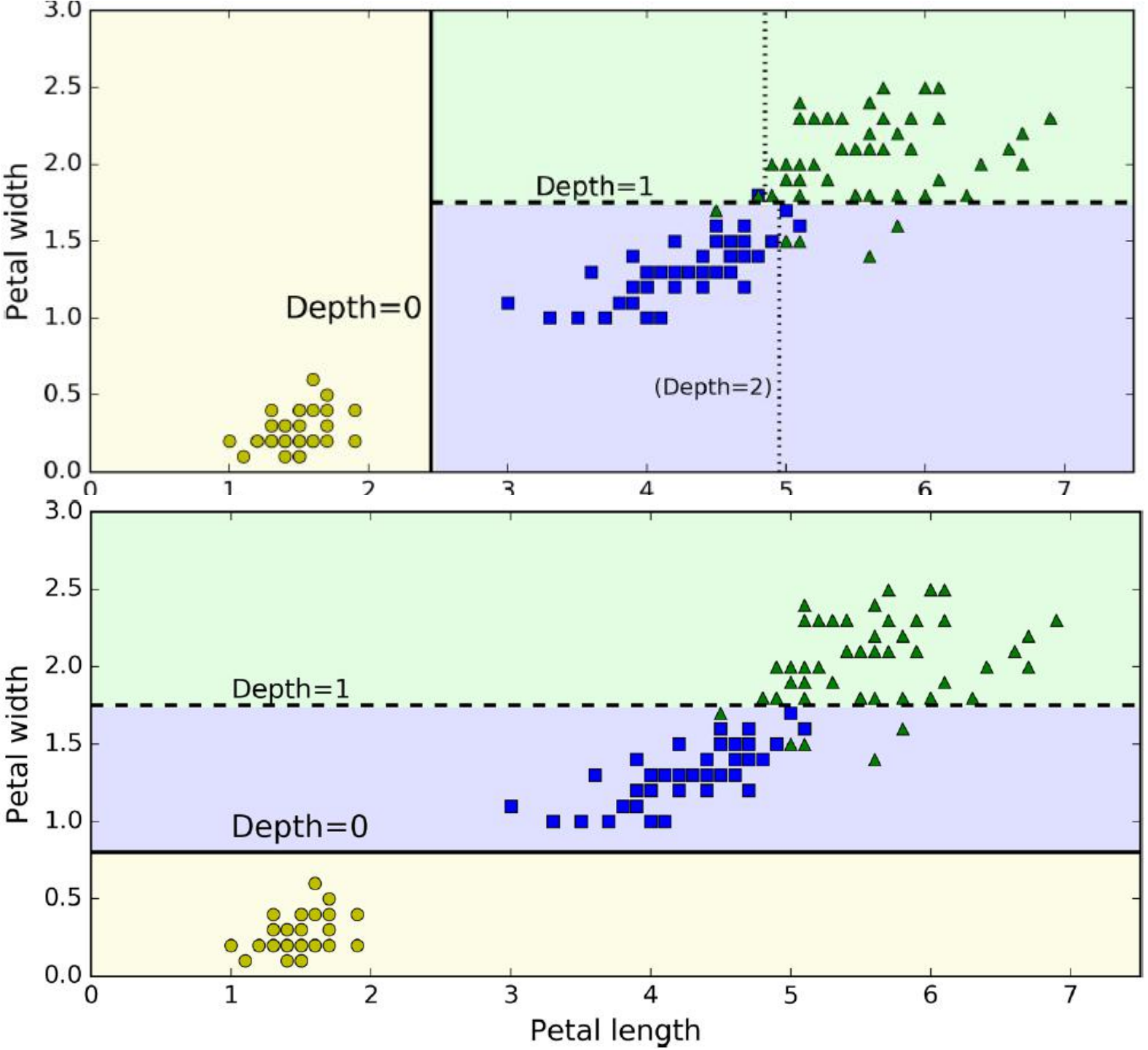
- 决策树很容易理解和解释，易于使用且功能丰富而强大。然而，它也有一些限制，首先，你可能已经注意到了，决策树很喜欢设定正交化的决策边界，（所有边界都是和某一个轴相垂直的），这使得它对训练数据集的旋转很敏感。



# 不稳定性

- 决策树的另一个主要问题是它对训练数据的微小变化非常敏感，举例来说，我们仅仅从鸢尾花训练数据中将最宽的 Iris-Versicolor 拿掉（花瓣长 4.8 厘米，宽 1.8 厘米），然后重新训练决策树模型，你可能就会得到非常不同的模型。事实上，由于 Scikit-Learn 的训练算法是非常随机的，即使是相同的训练数据你也可能得到差别很大的模型（除非你设置了随机数种子）。


Depth=1
Depth=0
(Depth=2)
Petal width
Depth=1
Depth=0
Petal width
Petal length

# 集成学习和随机森林

如果你随机向几千个人询问一个复杂问题，然后汇总他们的回答。在许多情况下，你会发现，这个汇总的回答比专家的回答还要好。这被称为群体智慧。同样，如果你聚合一组预测器（比如分类器或回归器）的预测，得到的预测结果也比最好的单个预测器要好。这样的一组预测器，我们称为集成，所以这种技术，也被称为集成学习。

# 集成学习和随机森林

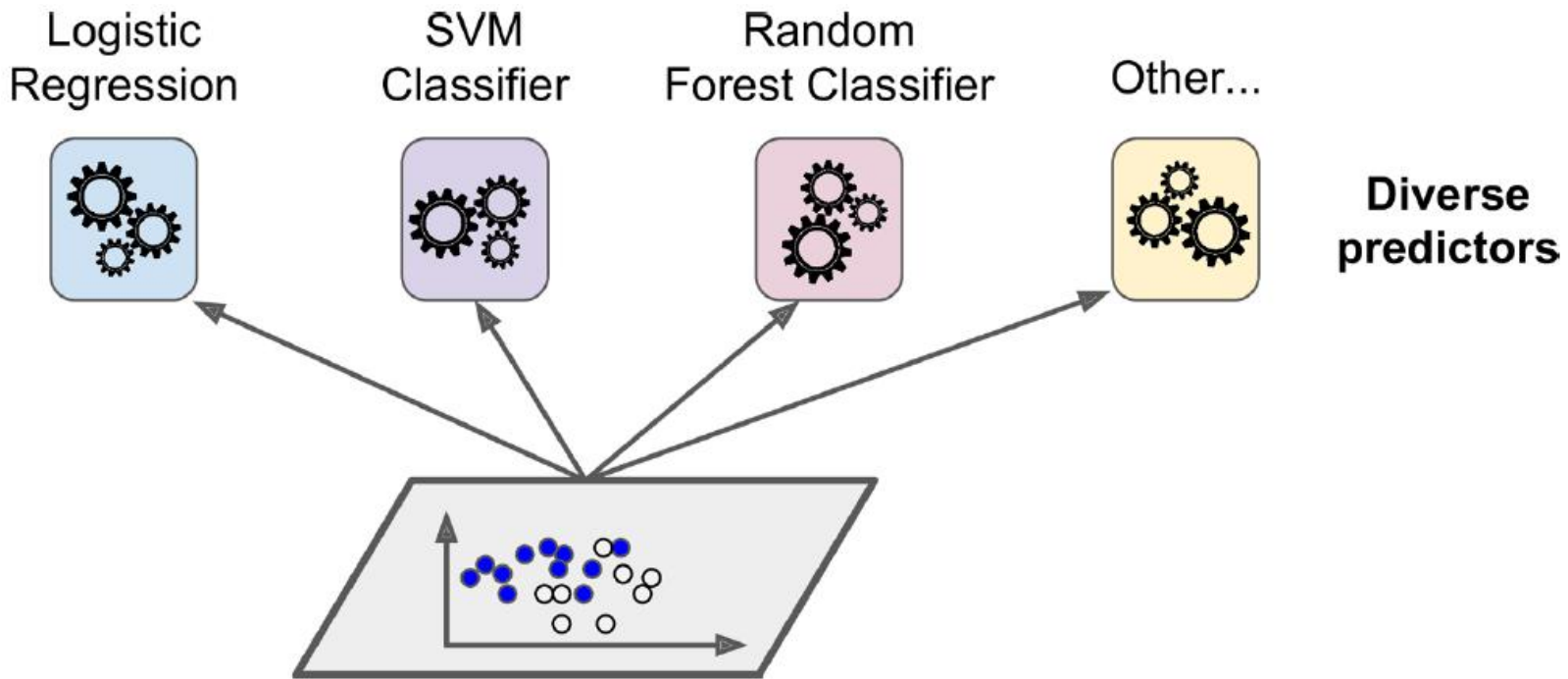
- 例如，可以训练一组决策树分类器，每一棵树都基于训练集不同的随机子集进行训练。做出预测时，只需要获得所有树各自的预测，然后给出得票最多的类别作为预测结果。
- 这样一组决策树的集成被称为**随机森林**，尽管很简单，但它是迄今可用的最强大的机器学习算法之一。

# 集成学习和随机森林

- 你可能已经构建好了一些不错的预测器，这时你就可以通过集成方法，将它们组合成一个更强的预测器。事实上，在机器学习竞赛中获胜的解决方案通常都涉及多种集成方法。
- 本章我们将探讨最流行的几种集成方法，包括bagging、boosting、stacking等，也将探索随机森林。

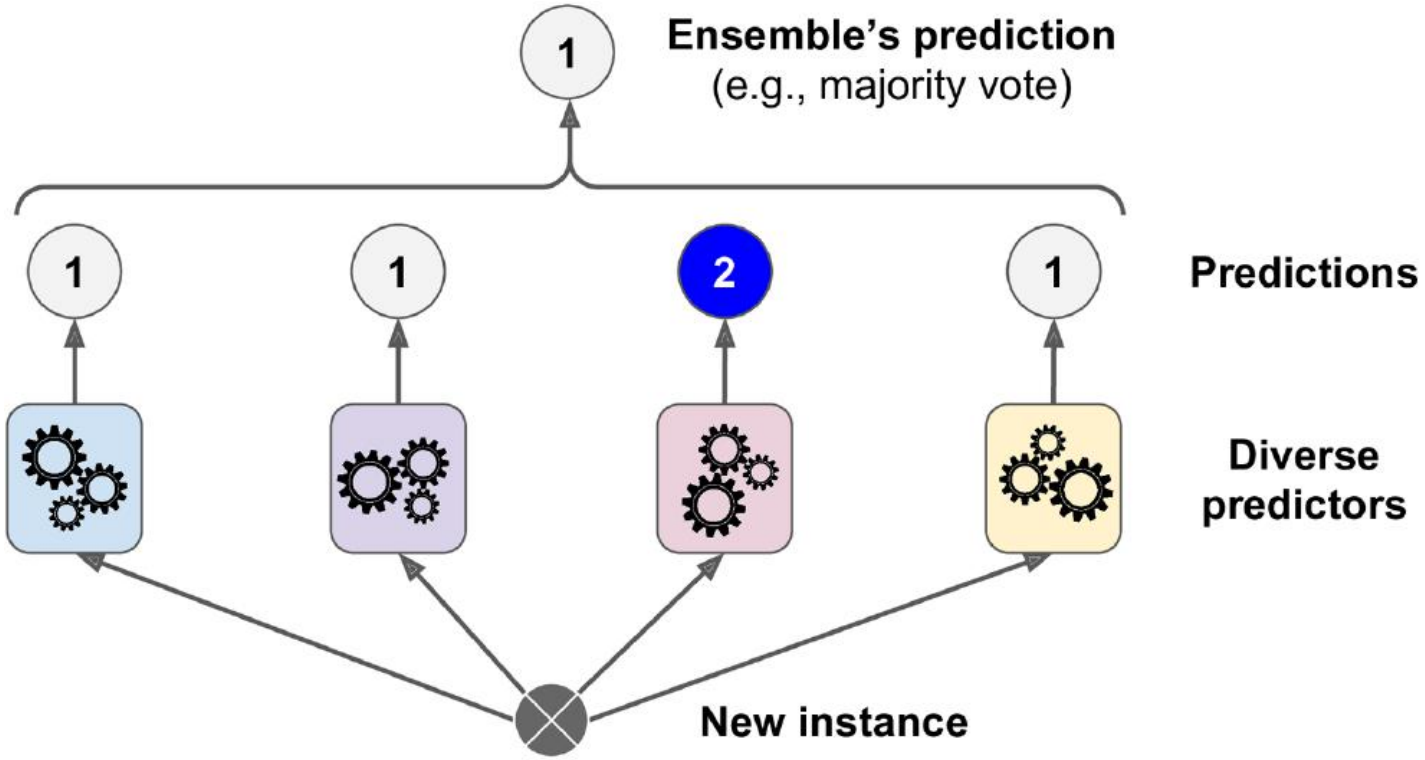
# 投票分类器

- 假设你已经训练好了一些分类器，每个分类器的准确率约为80%。大概包括：一个逻辑回归分类器、一个SVM分类器、一个随机森林分类器、一个K-近邻分类器，或许还有更多（见图）



# 投票分类器

- 要创建出一个更好的分类器，最简单的办法就是聚合每个分类器的预测，然后将得票最多的结果作为预测类别。这种大多数投票分类器被称为**硬投票分类器**（见图）。



# 投票分类器

- 这个投票法分类器的准确率通常比集成中最好的分类器还要高。事实上，即使每个分类器都是**弱学习器**（意味着它仅比随机猜测好一点），通过集成依然可以实现一个**强学习器**（高准确率），只要有足够大数量并且足够多种类的弱学习器就可以。

# 投票分类器

- 它们很可能会犯相同的错误，所以也会有很多次大多数投给了错误的类别，导致集成的准确率有所降低。
- 当预测器尽可能互相独立时，集成方法的效果最优。获得多种分类器的方法之一就是使用不同的算法进行训练。这会增加它们犯不同类型错误的机会，从而提升集成的准确率。

# 投票分类器

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
voting_clf = VotingClassifier(
        estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
        voting='hard'
    )
voting_clf.fit(X_train, y_train)
```

# 投票分类器

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
>>>     clf.fit(X_train, y_train)
>>>     y_pred = clf.predict(X_test)
>>>     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

LogisticRegression 0.864

RandomForestClassifier 0.872

SVC 0.888

VotingClassifier 0.896

# 投票分类器

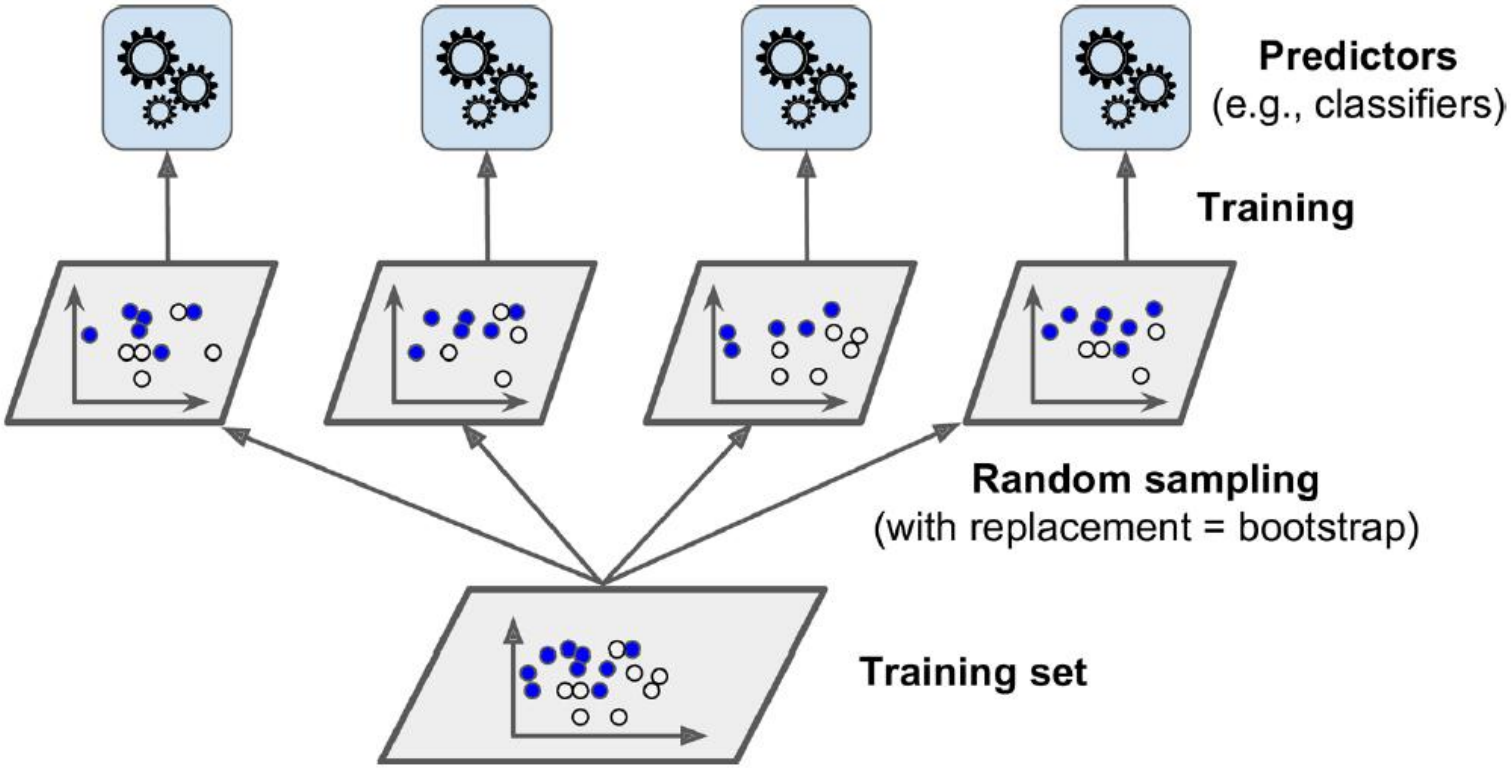
- 如果所有分类器都能够估算出类别的概率（即有predict_proba（）方法），那么你可以将概率在所有单个分类器上平均，然后让Scikit-Learn给出平均概率最高的类别作为预测。这被称为**软投票法**。通常来说，它比硬投票法的表现更优，因为它给予那些高度自信的投票更高的权重。

# Bagging和 Pasting

- 前面提到，获得不同种类分类器的方法之一是使用不同的训练算法。还有另一种方法是每个预测器使用的算法相同，但是在不同的训练集随机子集上进行训练。
- 采样时如果将样本放回，这种方法叫作bagging（bootstrap aggregating的缩写，也叫自举汇聚法）；采样时样本不放回，这种方法则叫作pasting。

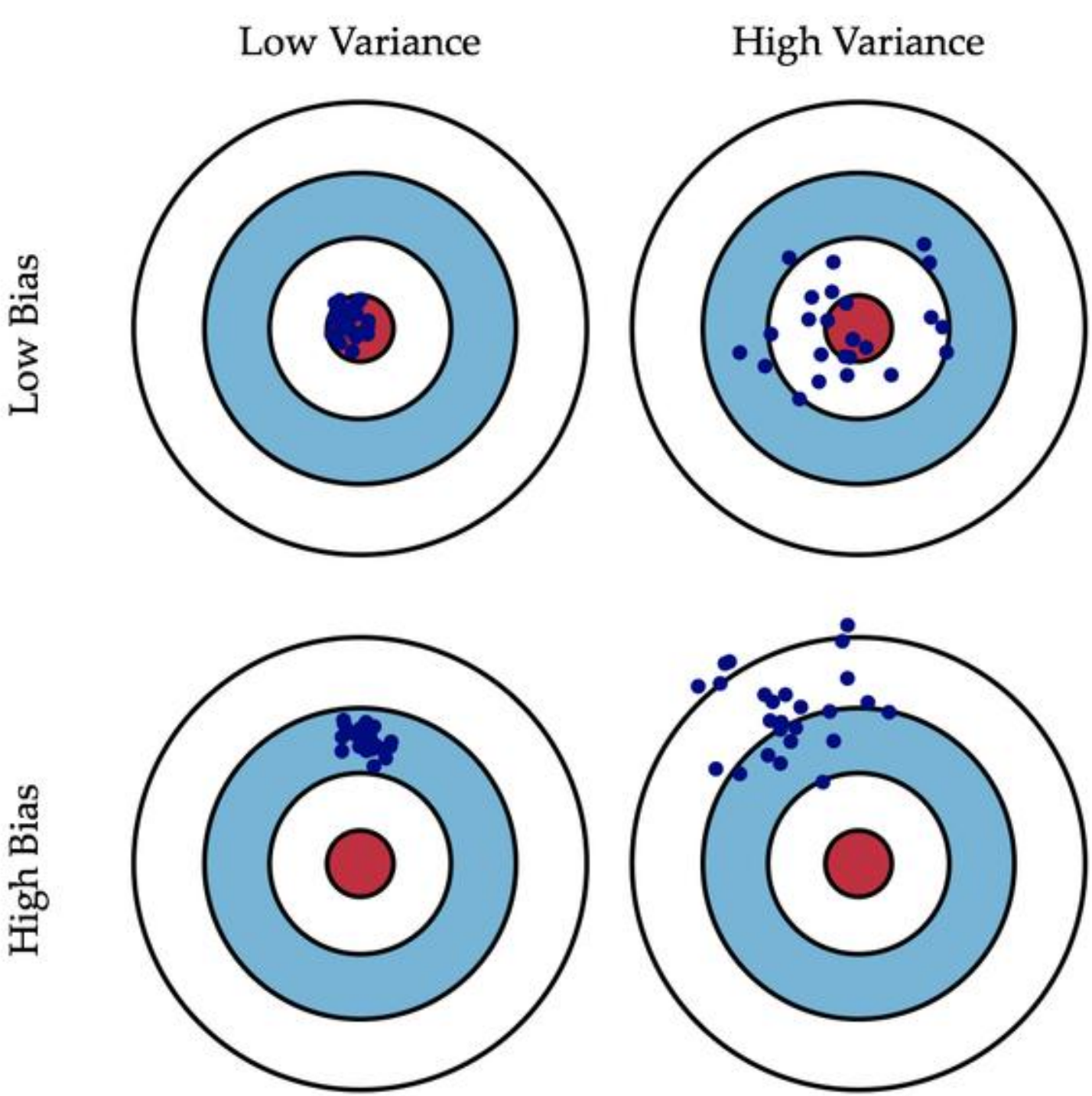
# Bagging和 Pasting

- bagging和pasting都允许训练实例在多个预测器中被多次采样，但是只有bagging允许训练实例被同一个预测器多次采样。采样过程和训练过程如图所示。



# Bagging和 Pasting

- 一旦预测器训练完成，集成就可以通过简单地聚合所有预测器的预测，来对新实例做出预测。聚合函数通常是统计法（即最多数的预测好比硬投票分类器一样）用于分类，或是平均法用于回归。
- 每个预测器单独的偏差都高于在原始训练集上训练的偏差，但是通过聚合，同时降低了偏差和方差。总体来说，最终结果是，与直接在原始训练集上训练的单个预测器相比，集成的偏差相近，但是方差更低


Low Variance
High Variance
Low Bias
High Bias

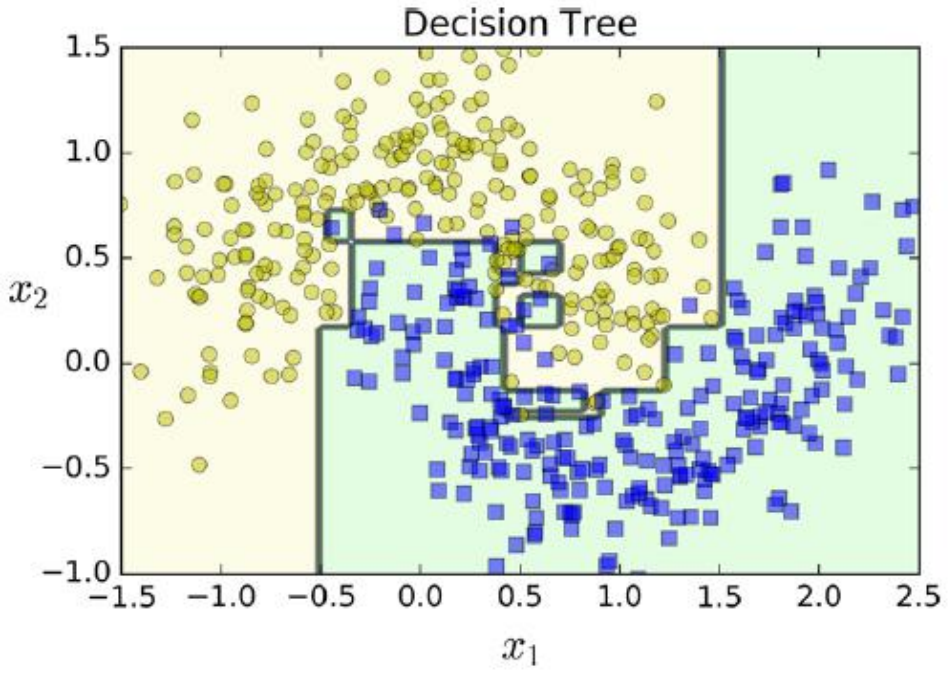
# Scikit-Learn的Bagging和 Pasting

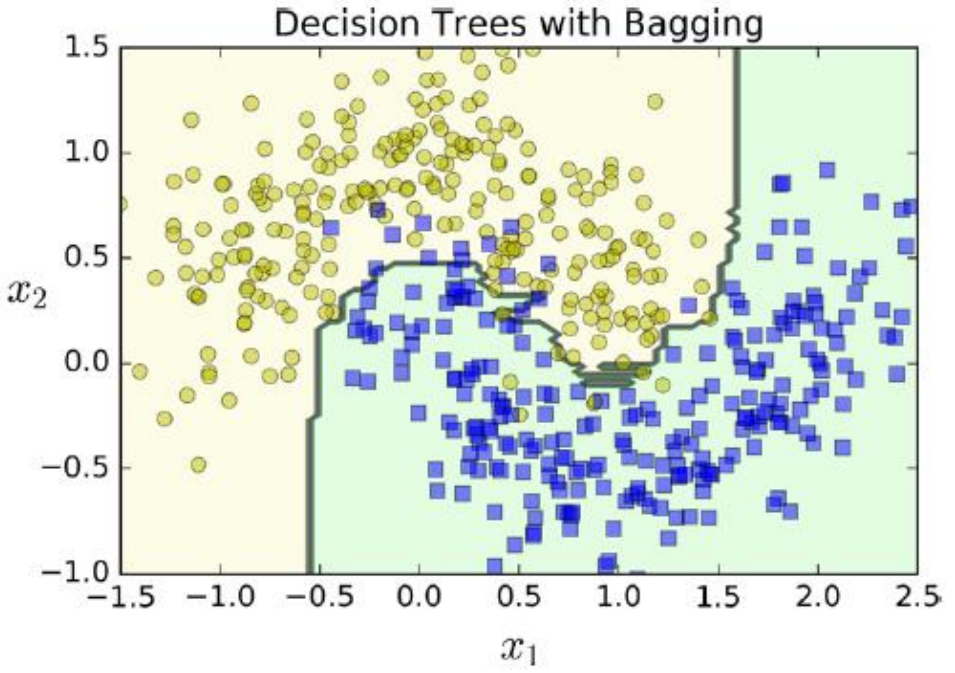
- Scikit-Learn提供了一个简单的API，可用BaggingClassifier类进行bagging和pasting（或BaggingRegressor用于回归）。
- 以下代码训练了一个包含500个决策树分类器的集成，每次随机从训练集中采样100个训练实例进行训练，然后放回（这是一个bagging的示例，如果你想使用pasting，只需要设置bootstrap=False即可）。参数n_jobs用来指示Scikit-Learn用多少CPU内核进行训练和预测（-1表示让Scikit-Learn使用所有可用内核）：

# Scikit-Learn的Bagging和 Pasting

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
bag_clf = BaggingClassifier(
        DecisionTreeClassifier(), n_estimators=500,
        max_samples=100, bootstrap=True, n_jobs=-1
    )
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

*A single Decision Tree versus a bagging ensemble of 500 trees*



# Scikit-Learn的Bagging和 Pasting

- 由于自助法给每个预测器的训练子集引入了更高的多样性，所以最后bagging比pasting的偏差略高，但这也意味着预测器之间的关联度更低，集成的方差降低。
- 总之，bagging生成的模型通常更好，这也就是为什么它更受欢迎。但是，如果你有充足的时间和CPU资源，可以使用交叉验证来对bagging和pasting的结果进行评估，再做出最合适的选择。

# 包外评估 Out-of-Bag Evaluation

- 对于任意给定的预测器，使用bagging，有些实例可能会被采样多次，而有些实例则可能根本不被采样。BaggingClassifier默认采样m个训练实例，然后放回样本（bootstrap=True），m是训练集的大小。这意味着对于每个预测器来说，平均只对63%的训练实例进行采样。剩余37%未被采样的训练实例称为包外（out of bag）实例。注意，对其他预测器来说，这是不一样的37%
- 既然预测器在训练的时候从未见过这些包外实例，正好可以用这些实例进行评估，从而不需要单独的验证集或是交叉验证。

# 包外评估

- Scikit-Learn中，创建BaggingClassifier时，设置oob_score=True，就可以请求在训练结束后自动进行包外评估。
- 下面的代码演示了这一点。通过变量 oob_score_ 可以得到最终的评估分数：

```
>>> bag_clf = BaggingClassifier(
>>>         DecisionTreeClassifier(), n_estimators=500,
>>>         bootstrap=True, n_jobs=-1, oob_score=True)
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9306666666666664
```

# 包外评估

```
>>> bag_clf = BaggingClassifier(
>>>         DecisionTreeClassifier(), n_estimators=500,
>>>         bootstrap=True, n_jobs=-1, oob_score=True)
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9306666666666664

>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9360000000000005
```

# 包外评估

- 每个训练实例的包外决策函数也可以通过变量oob_decision_function_获得。本例中（基础预测器具备predict_proba（）方法），决策函数返回的是每个实例的类别概率。例如，包外评估估计，第二个训练实例有60.6%的概率属于正类（以及39.4%的概率属于负类）

```
>>> bag_clf.oob_decision_function_
array([[ 0.        ,  1.        ],
       [ 0.60588235,  0.39411765],
       [ 1.        ,  0.        ],
       ...
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.48958333,  0.51041667]])
```

# Random Patches和随机子空间

- BaggingClassifier也支持对特征进行抽样，这通过两个超参数控制：max_features和bootstrap_features。它们的工作方式跟max_samples和bootstrap相同，只是抽样对象不再是实例，而是特征。因此，每个预测器将用输入特征的随机子集进行训练。
- 这对于处理高维输入（例如图像）特别有用。对训练实例和特征都进行抽样，被称为**Random Patches方法**。

# Random Patches和随机子空间

- 而保留所有训练实例（即bootstrap=False并且max_samples=1.0）但是对特征进行抽样（即bootstrap_features=True并且/或max_features<1.0），这被称为**随机子空间法**。对特征抽样给预测器带来更大的多样性，所以以略高一点的偏差换取了更低的方差。

# 随机森林

- 前面已经提到，随机森林是决策树的集成，通常用bagging（有时也可能是pasting）方法训练，训练集大小通过max_samples来设置。除了先构建一个BaggingClassifier然后将结果传输到DecisionTreeClassifier，还有一种方法就是使用RandomForestClassifier类，
- 以下代码使用所有可用的CPU内核，训练了一个拥有500棵树的随机森林分类器（每棵树限制为最多16个叶节点）：

# 随机森林

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

# 随机森林

- 随机森林在树的生长上引入了更多的随机性：分裂节点时不再是搜索最好的特征，而是在一个随机生成的特征子集里搜索最好的特征。这导致决策树具有更大的多样性，（再一次）用更高的偏差换取更低的方差，总之，还是产生了一个整体性能更优的模型。
- 下面的BaggingClassifier大致与前面的RandomForestClassifier相同：

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1
)
```

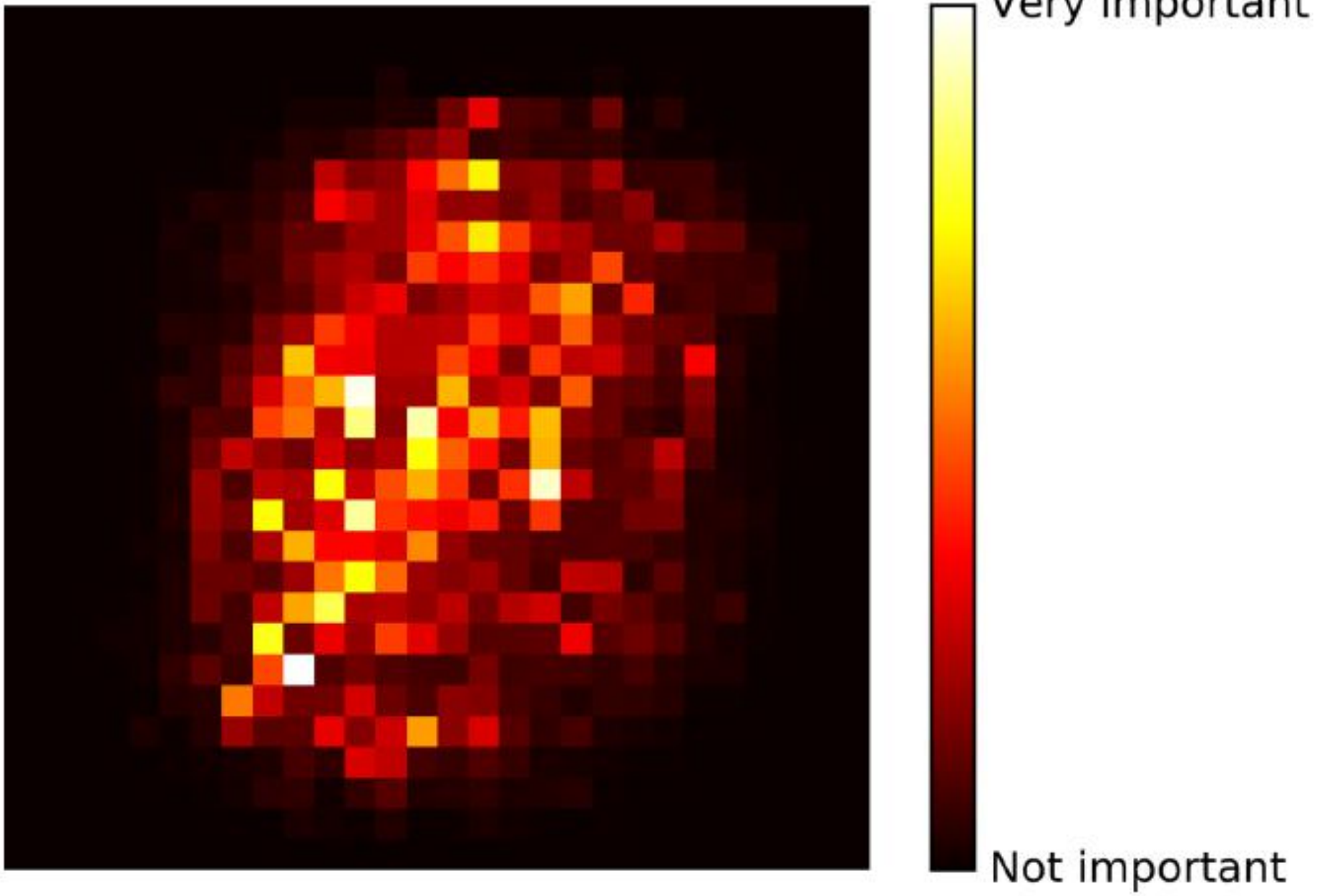
# 特征重要性

- 如果你查看单个决策树会发现，重要的特征更可能出现在靠近根节点的位置，而不重要的特征通常出现在靠近叶节点的位置（甚至根本不出现）。因此，通过计算一个特征在森林中所有树上的平均深度，可以估算出一个特征的重要程度。
- Scikit-Learn在训练结束后自动计算每个特征的重要性。通过变量feature_importances_你就可以访问到这个计算结果。

# 特征重要性

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
>>>         print(name, score)
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

*MNIST pixel importance (according to a Random Forest classifier)*



# 极端随机树 Extra-Trees

- 随机森林里单棵树的生长过程中，每个节点在分裂时仅考虑到了一个随机子集所包含的特征。如果我们对每个特征使用随机阈值，而不是搜索得出的最佳阈值（如常规决策树），则可能让决策树生长得更加随机。
- 这种极端随机的决策树组成的森林，被称为极端随机树集成（或简称Extra-Trees）。同样，它也是以更高的偏差换取了更低的方差。极端随机树训练起来比常规随机森林要快很多，因为在每个节点上找到每个特征的最佳阈值是决策树生长中最耗时的任务之一。
- 使用Scikit-Learn的ExtraTreesClassifier可以创建一个极端随机树分类器。它的API与RandomForestClassifier相

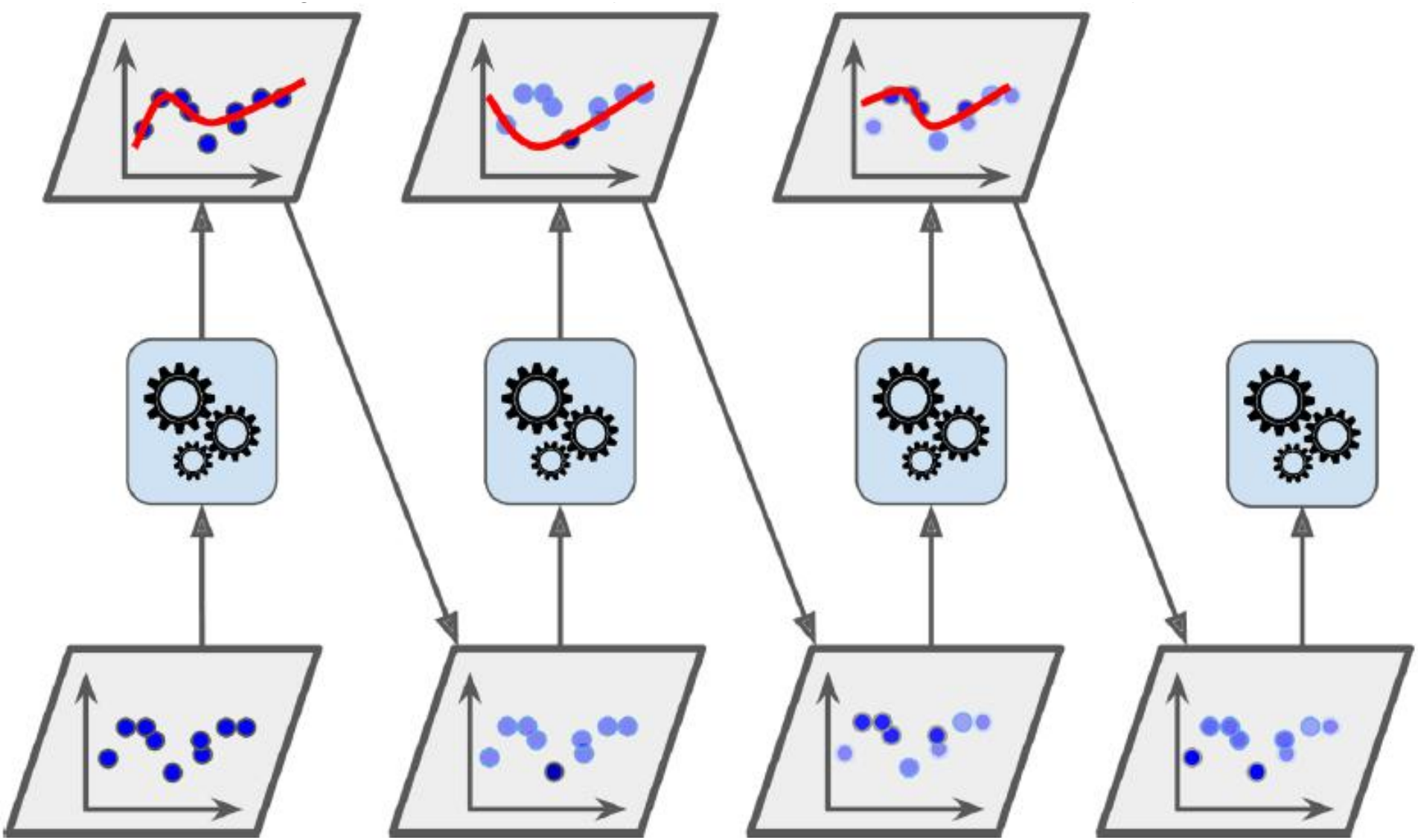
# 提升法 Boosting

- 提升法（Boosting）是指可以将几个弱学习器结合成一个强学习器的任意集成方法。大多数提升法的总体思路是循环训练预测器，每一次都对其前序做出一些改正。可用的提升法有很多，但目前最流行的方法是AdaBoost（自适应提升法，Adaptive Boosting的缩写）和梯度提升。我们先从AdaBoost开始介绍。

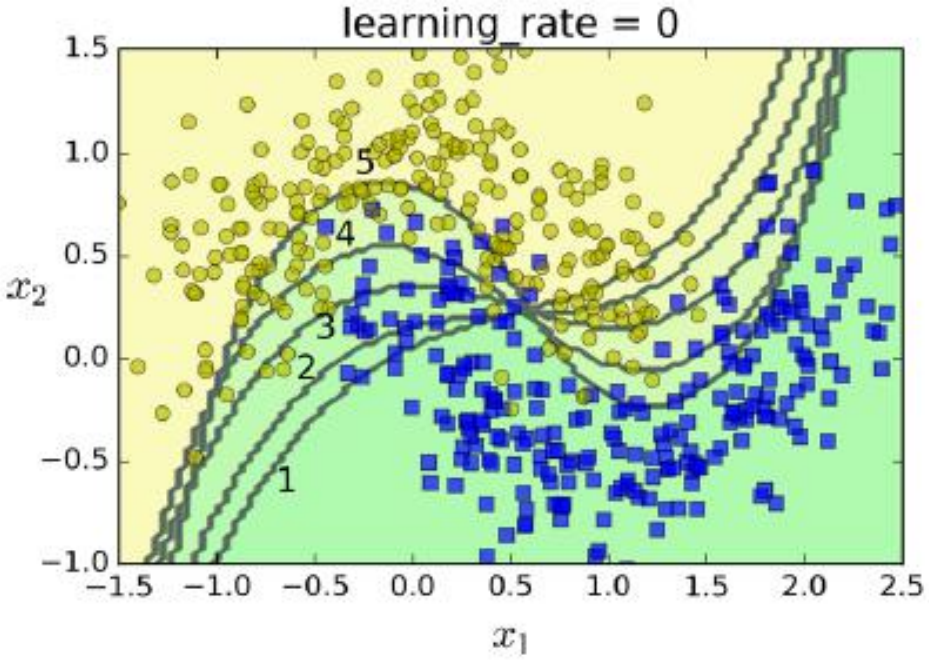
# AdaBoost

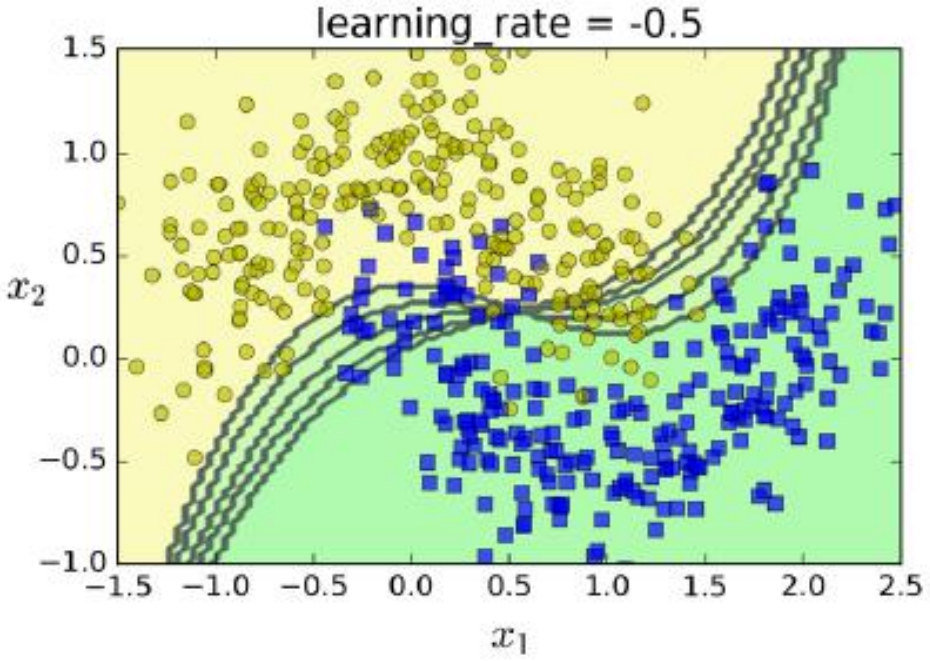
- 新预测器对其前序进行纠正的办法之一，就是更多地关注前序拟合不足的训练实例。从而使新的预测器不断地越来越专注于难缠的问题，这就是AdaBoost使用的技术。
- 例如，要构建一个AdaBoost分类器，首先需要训练一个基础分类器（比如决策树），用它对训练集进行预测。然后对错误分类的训练实例增加其相对权重，接着，使用这个最新的权重对第二个分类器进行训练，然后再次对训练集进行预测，继续更新权重，并不断循环向前。

# AdaBoost



# *Decision boundaries of consecutive predictors*



# AdaBoost

- 我们来仔细看看AdaBoost算法。每个实例的权重$w^{(i)}$ 最初设置为 1/m。第一个预测器训练后，计算其加权误差率$r_1$，见公式7-1

*Equation 7-1. Weighted error rate of the $j^{th}$ predictor*

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

where $\hat{y}_j^{(i)}$ is the $j^{th}$ predictor's prediction for the $i^{th}$ instance.

# AdaBoost

- 预测器的权重$\alpha_j$通过公式7-2来计算，其中η是学习率超参数（默认为1）。预测器的准确率越高，其权重就越高。如果它只是随机猜测，则其权重接近于零。但是，如果大部分情况下它都是错的（也就是准确率比随机猜测还低），那么它的权重为负。

*Equation 7-2. Predictor weight*

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

# AdaBoost

- 接下来，使用公式7-3，对实例的权重进行更新，也就是提升被错误分类的实例的权重。

*Equation 7-3. Weight update rule*

$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

# AdaBoost

- 最后，使用更新后的权重训练一个新的预测器，然后重复整个过程（计算新预测器的权重，更新实例权重，然后对另一个预测器进行训练，等等）。当到达所需数量的预测器，或得到完美的预测器时，算法停止。
- 预测的时候，AdaBoost就是简单地计算所有预测器的预测结果，并使用预测器权重$\alpha_j$对它们进行加权。最后，得到大多数加权投票的类别就是预测器给出的预测类别（见公式7-4）

*Equation 7-4. AdaBoost predictions*

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

# AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
```

- ada_clf.fit(X_train, y_train)

Scikit-Learn使用的其实是AdaBoost的一个多分类版本，叫作SAMME（基于多类指数损失函数的逐步添加模型）。当只有两个类别时，SAMME即等同于AdaBoost。此外，如果预测器可以估算类别概率，Scikit-Learn会使用一种SAMME的变体，称为SAMME.R（R代表“Real”），它依赖的是类别概率而不是类别预测，通常表现更好。

# 梯度提升 Gradient Boosting

- 一个非常受欢迎的提升法是梯度提升（Gradient Boosting）。跟AdaBoost一样，梯度提升也是逐步在集成中添加预测器，每一个都对其前序做出改正。不同之处在于，它不是像AdaBoost那样在每个迭代中调整实例权重，而是让新的预测器针对前一个预测器的残差进行拟合。

# 梯度提升 Gradient Boosting

- 我们来看一个简单的回归示例，使用决策树作为基础预测器（梯度提升当然也适用于回归任务），这被称为梯度树提升或者是梯度提升回归树（GBRT）。首先，在训练集（比如带噪声的二次训练集）上拟合一个DecisionTreeRegressor：

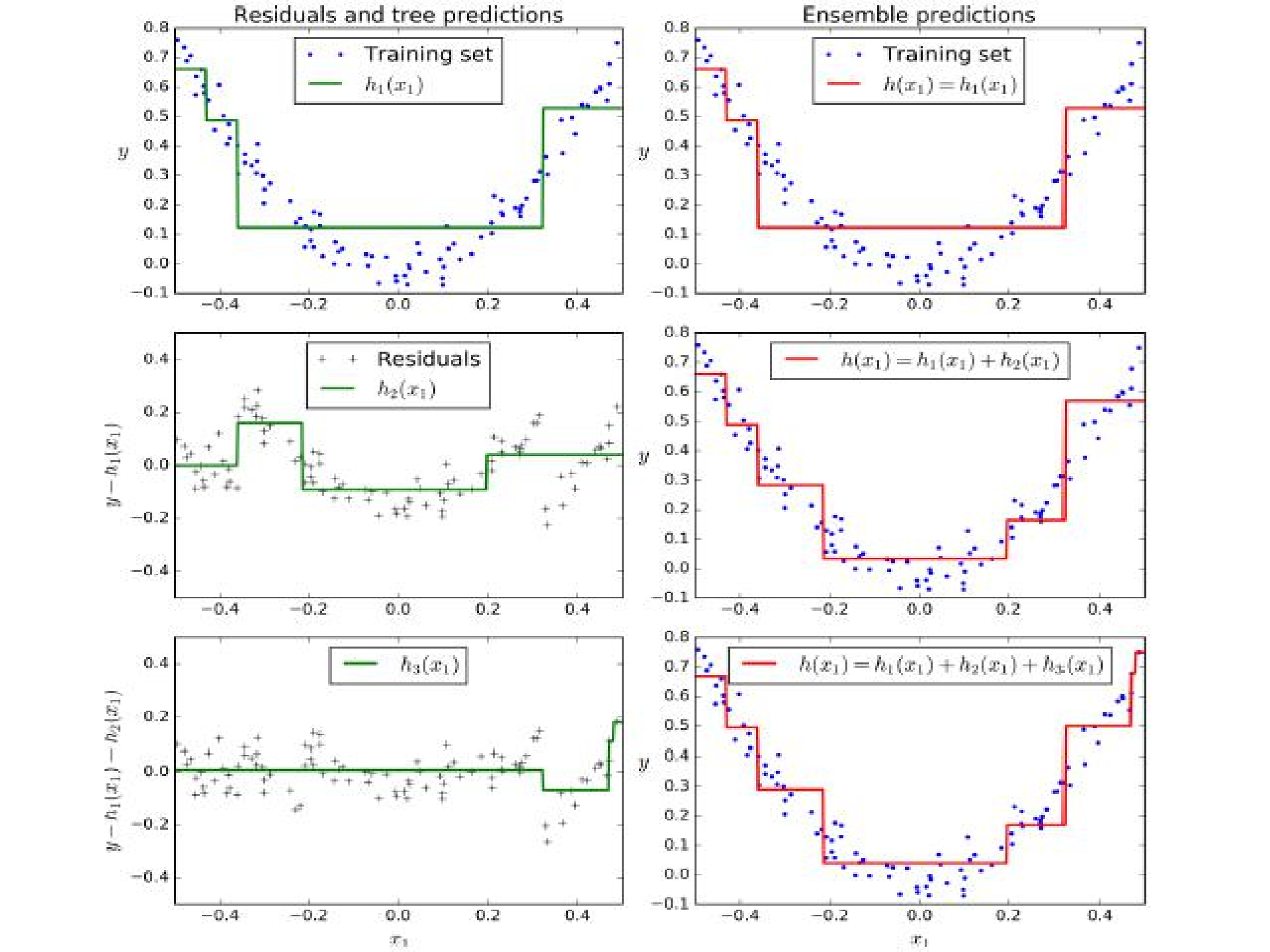
# 梯度提升 Gradient Boosting

```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)

y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

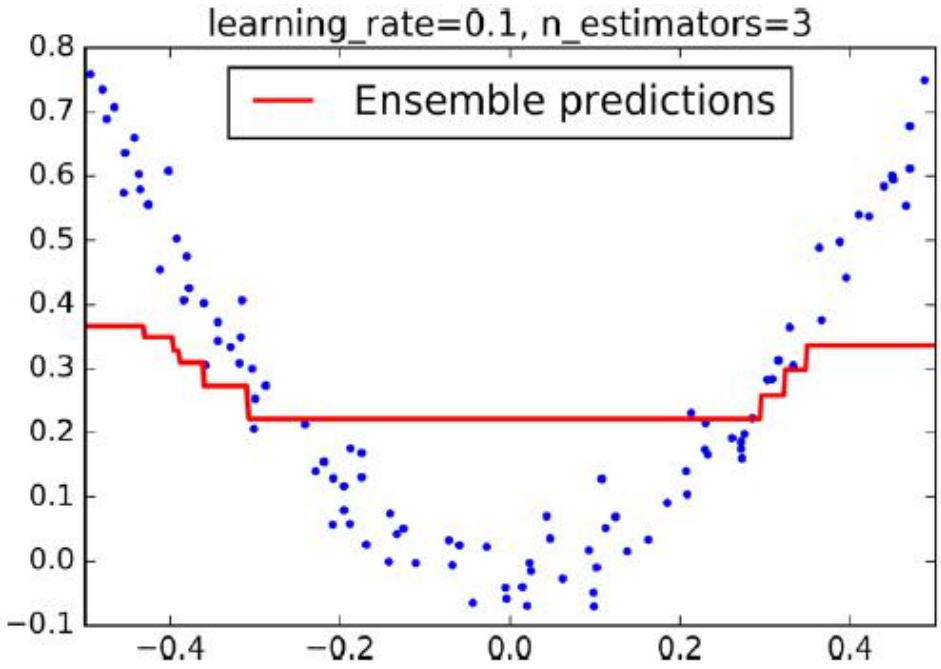

Residuals and tree predictions
Ensemble predictions
Training set
$h_1(x_1)$
$h(x_1) = h_1(x_1)$
Residuals
$h_2(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1)$
$h_3(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1) + h_3(x_1)$
$y$
$y - h_1(x_1)$
$y - h_1(x_1) - h_2(x_1)$
$x_1$

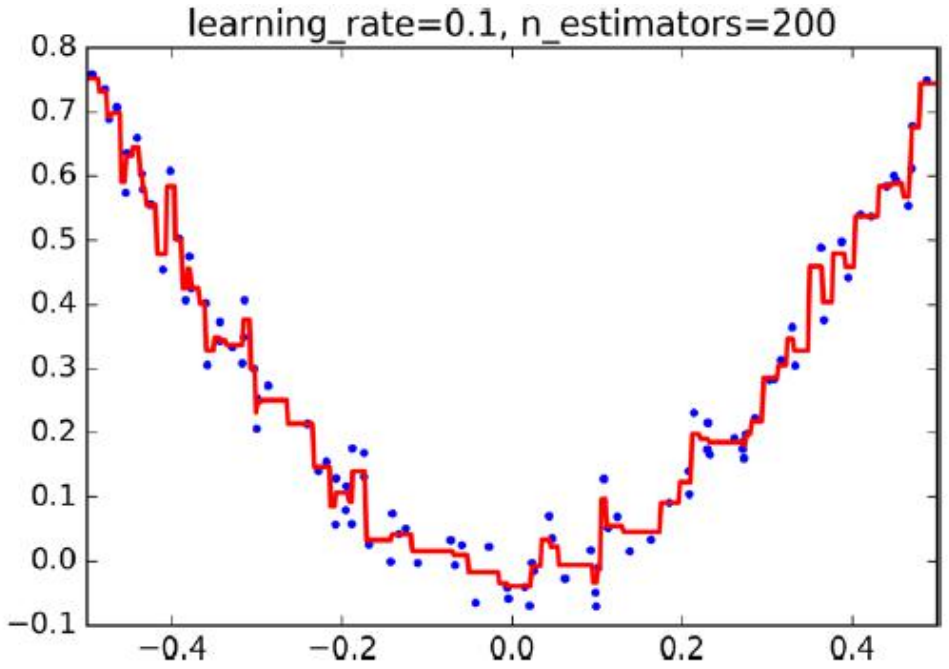
# 梯度提升 Gradient Boosting

**from sklearn.ensemble import** GradientBoostingRegressor

**gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)**

gbrt.fit(X, y)



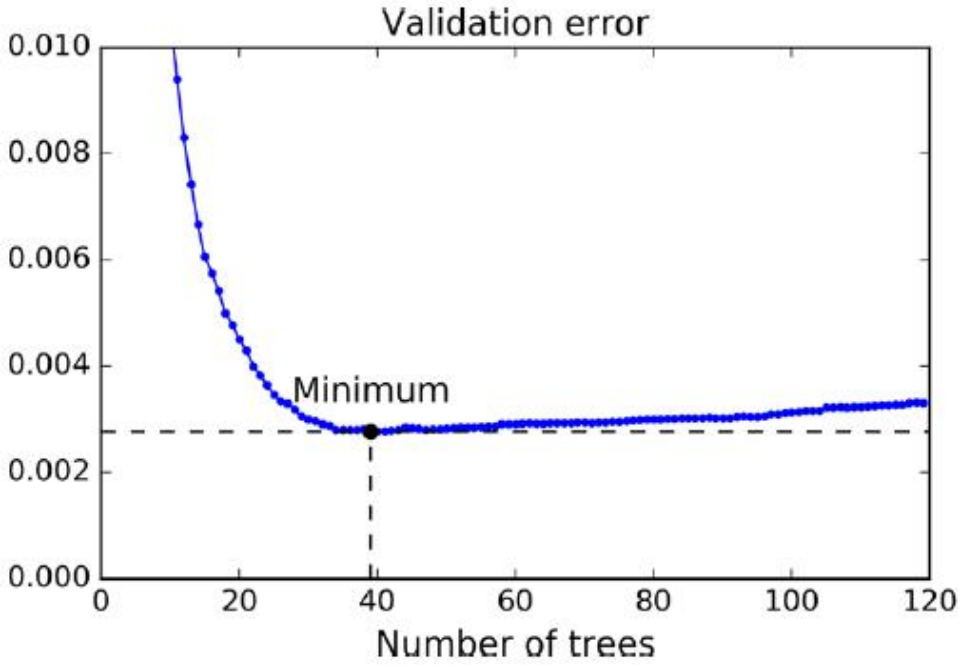



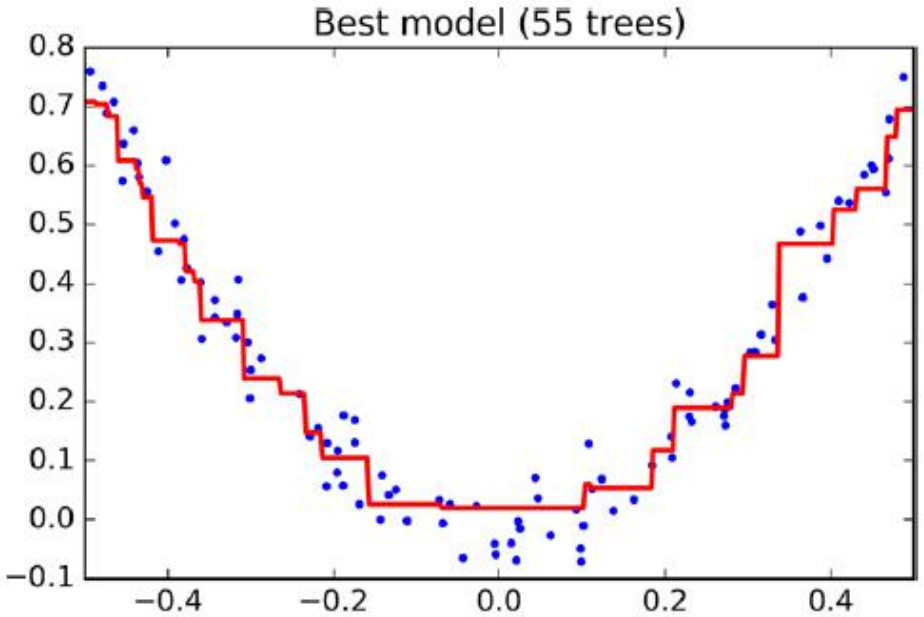
# 梯度提升 Gradient Boosting

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
X_train, X_val, y_train, y_val = train_test_split(X, y)
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)
errors = [mean_squared_error(y_val, y_pred)
                    for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)
gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

简单的实现方法就是使用 staged_predict（）方法：它在训练的每个阶段（一棵树时，两棵树时等等）都对集成的预测返回迭代器。

# *Tuning the number of trees using early stopping*



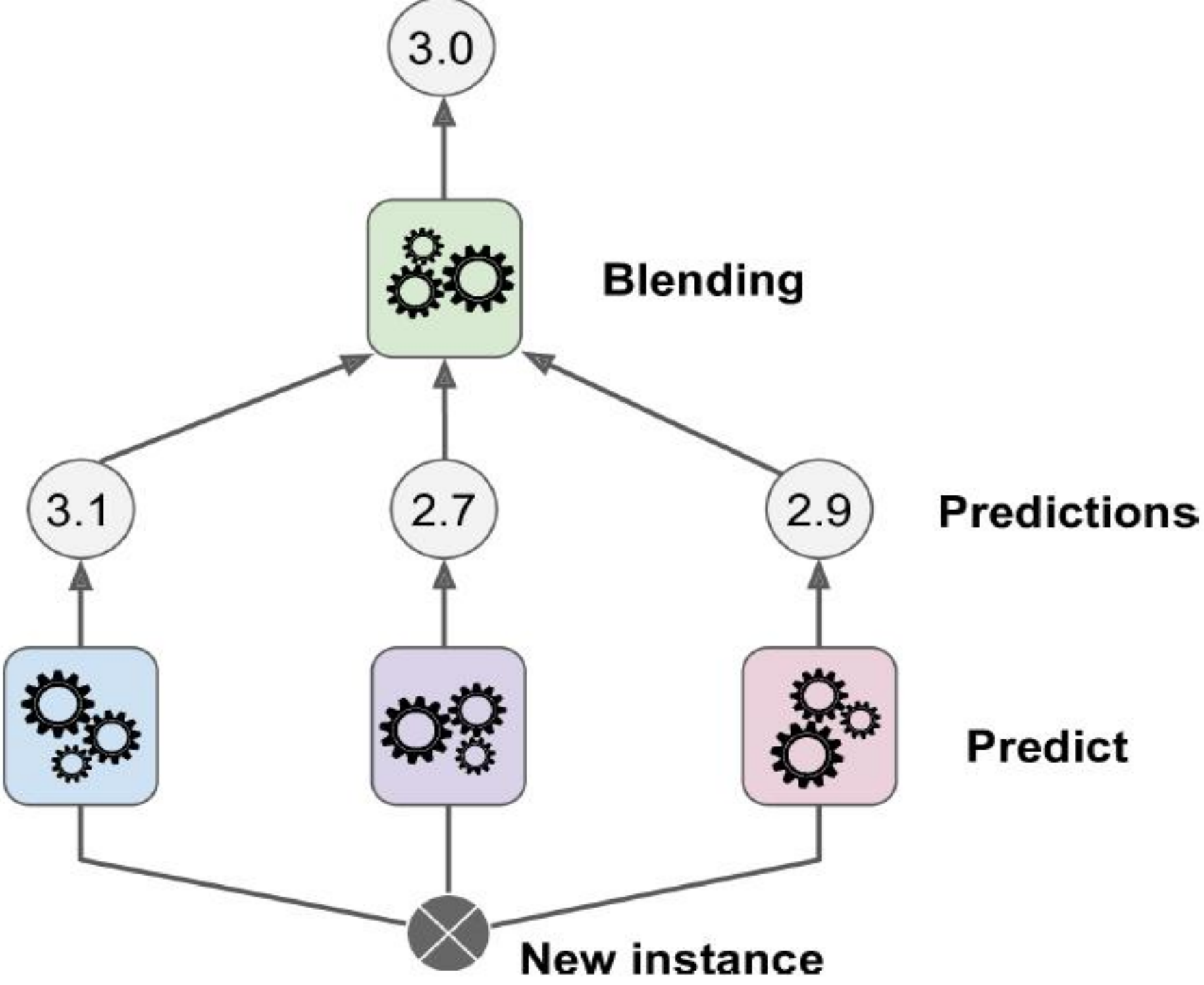



# 梯度提升 Gradient Boosting

- GradientBoostingRegressor类还可以支持超参数subsample，指定用于训练每棵树的实例的比例。例如，如果subsample=0.25，则每棵树用25%的随机选择的实例进行训练。现在你可以猜到，这也是用更高的偏差换取了更低的方差，同时在相当大的程度上加速了训练过程。这种技术被称为**随机梯度提升**。

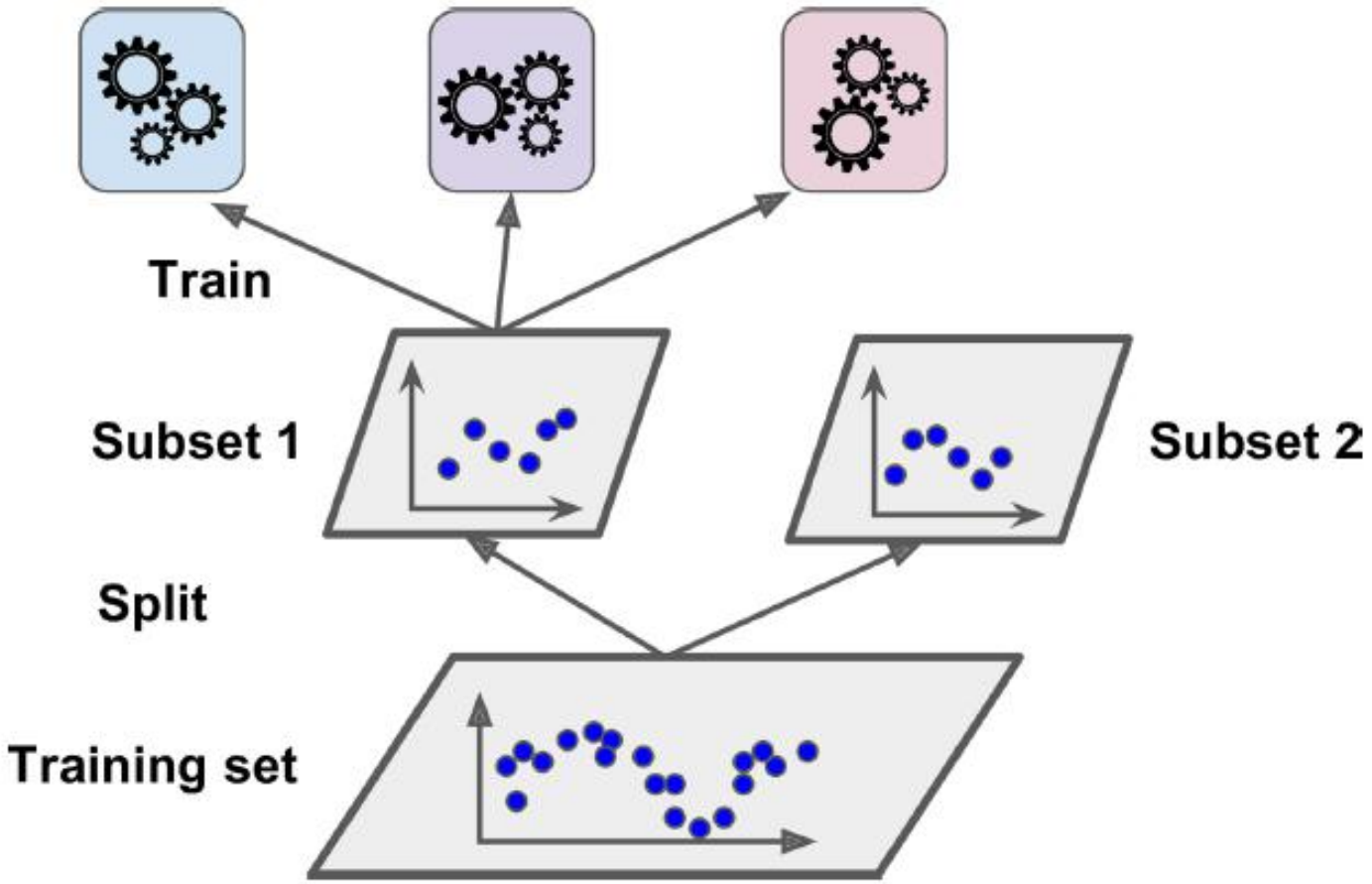
# 堆叠法 Stacking

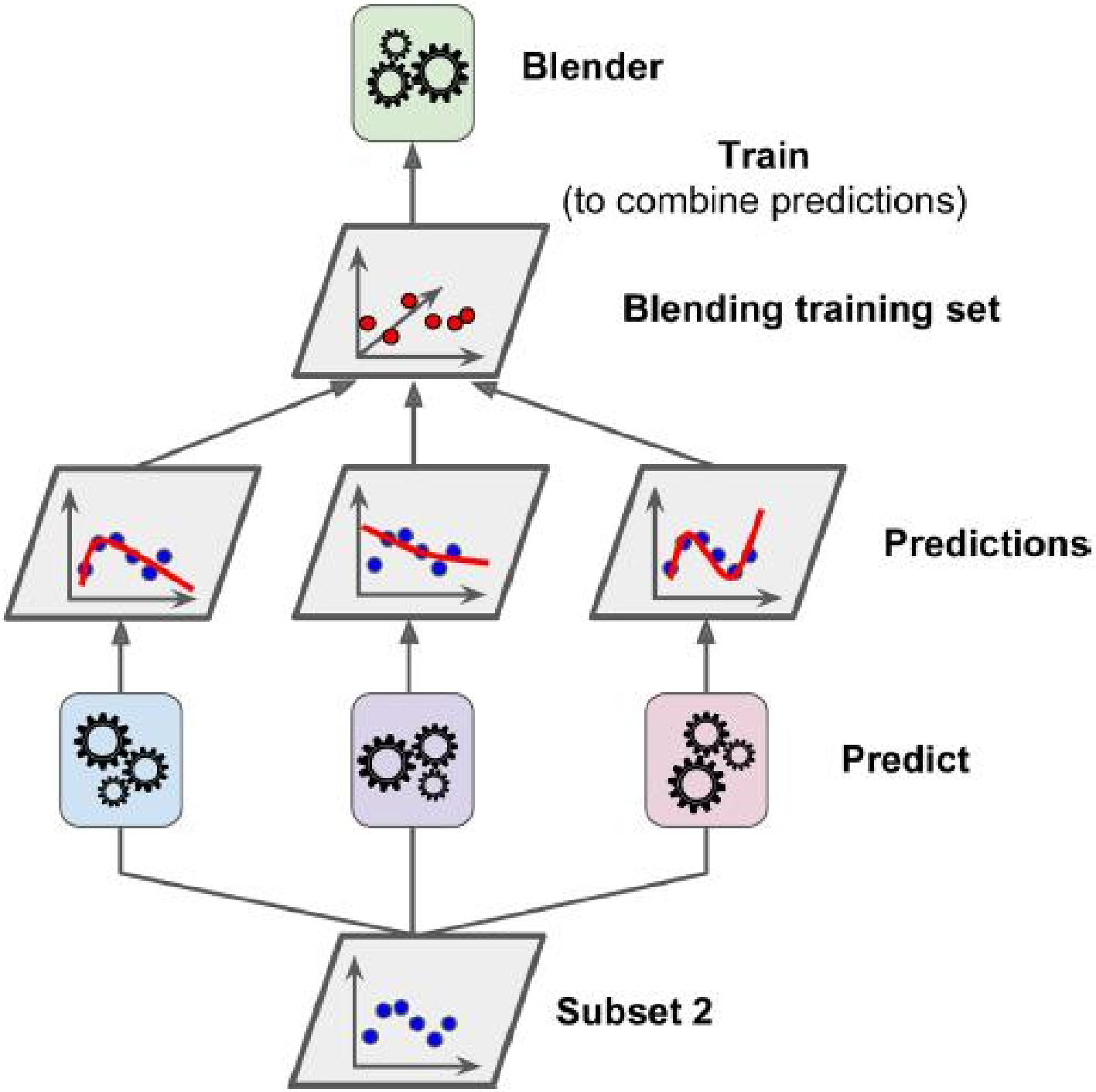
- 我们要讨论的最后一个集成方法叫作堆叠法（stacking），又称层叠泛化法。它基于一个简单的想法：与其使用一些简单的函数（比如硬投票）来聚合集成中所有预测器的预测，我们为什么不训练一个模型来执行这个聚合呢？图7-12显示了在新实例上执行回归任务的这样一个集成。底部的三个预测器分别预测了不同的值（3.1、2.7和2.9），然后最终的预测器（称为混合器或元学习器）将这些预测作为输入，进行最终预测（3.0）。


3.0
Blending
3.1
2.7
2.9
Predictions
Predict
New instance

# 堆叠法 Stacking

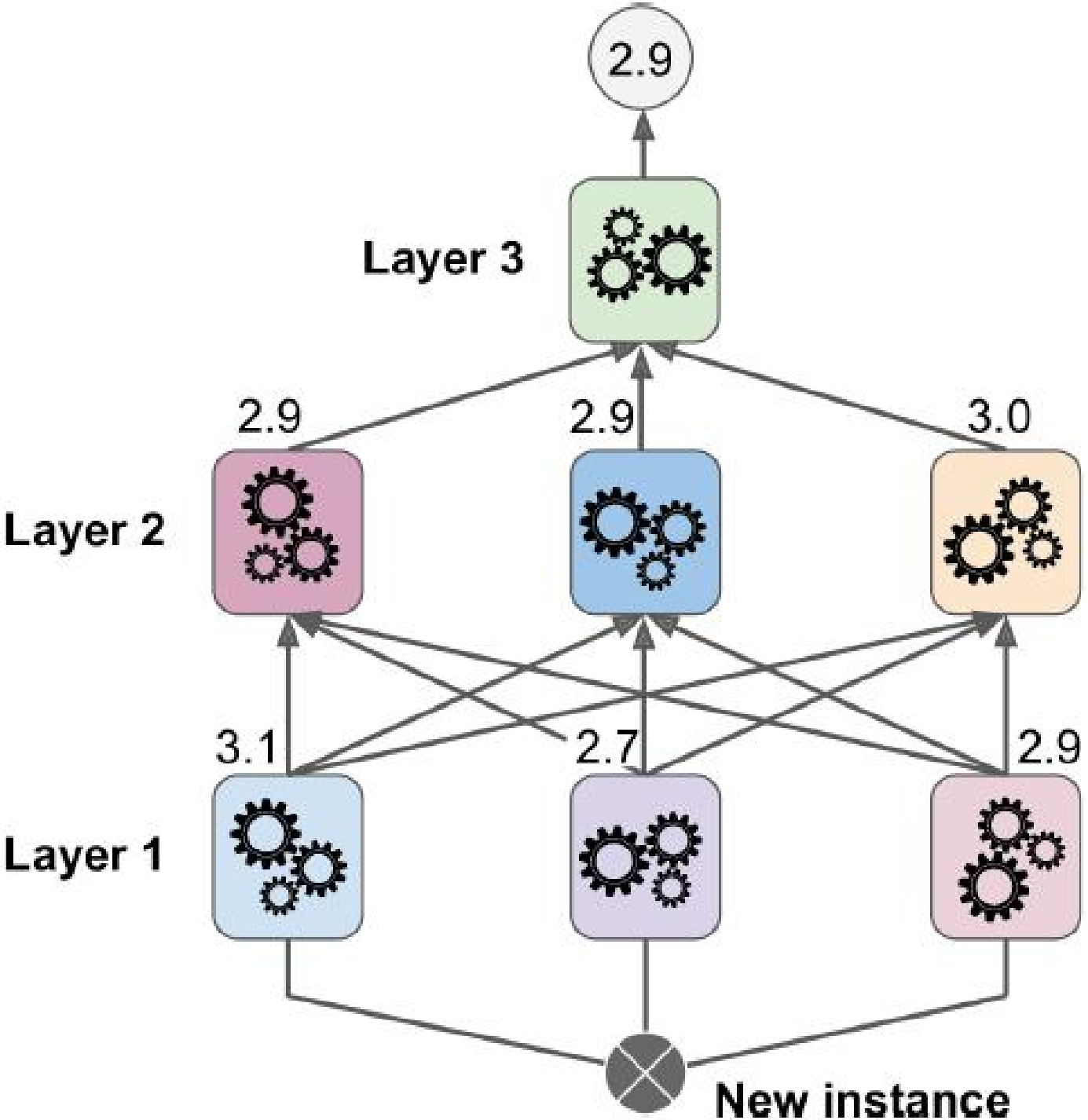
- 训练混合器的常用方法是使用留存集。我们看看它是如何工作的。首先，将训练集分为两个子集，第一个子集用来训练第一层的预测器（见图）。




Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2

# 堆叠法 Stacking

- 事实上，通过这种方法可以训练多种不同的混合器（例如，一个使用线性回归，另一个使用随机森林回归，等等）：于是我们可以得到一个混合器层。
- 诀窍在于将训练集分为三个子集：第一个用来训练第一层，第二个用来创造训练第二层的新训练集（使用第一层的预测），而第三个用来创造训练第三层的新训练集（使用第二层的预测）。一旦训练完成，我们可以按照顺序遍历每层来对新实例进行预测。


2.9
Layer 3
2.9
2.9
3.0
Layer 2
3.1
2.7
2.9
Layer 1
New instance