决策树和集成算法

2020 年 4 月 29 日

目录

1 决策树

决策树的动机很简单。就是一系列的选择（if，else），类似人做决策。但是如何做决策，提出什么问题，怎么回答这些，决策树则是基于对数据的学习。比如，你要建一套规则来识别自然界的某种动物，你可能会提出如下问题：

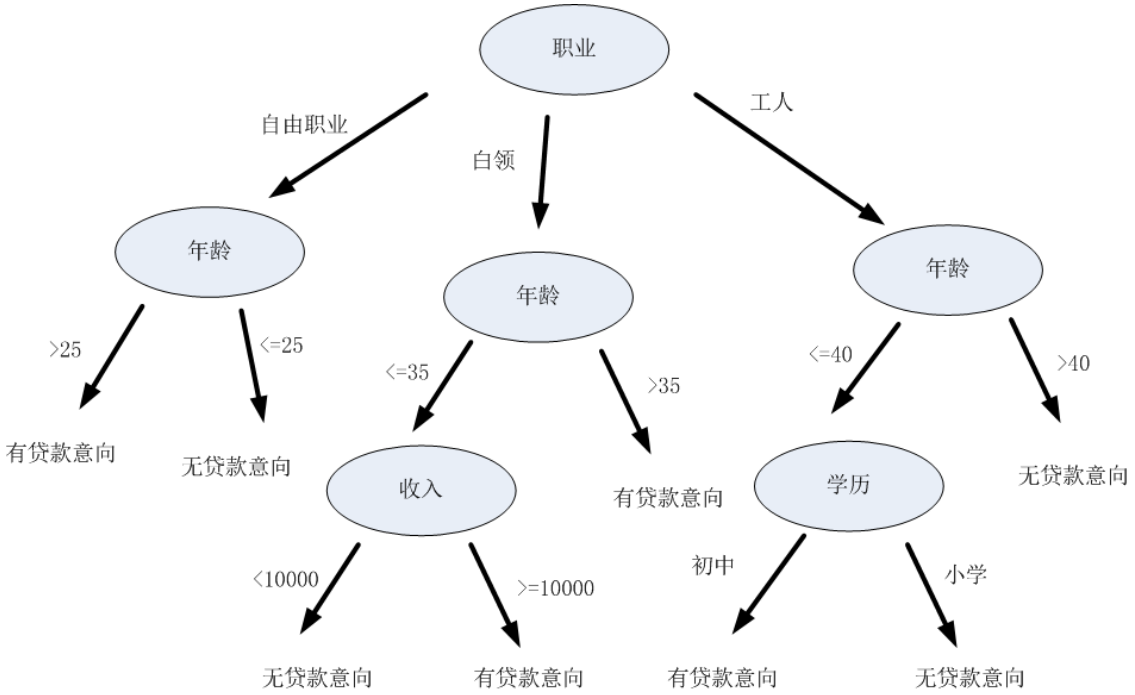
- 这个动物体长有一米吗？
- 比一米长：这个动物有角吗？
- 有：这个角比 10 厘米长吗？
- 没有：这些动物会佩戴项圈吗？
- 比一米短：这个动物两条腿还是四条腿？
- 两条：有翅膀吗？
- 四条：有毛茸茸的尾巴吗？

而如何提出问题来进行分裂，是决策树的关键（考虑：一个好问题直接给出答案？）

基于树的模型的一大优点是，极少的数据预处理（或者对数据的形态要求极少）。而且可以应付各类数据（离散的，连续的），而且不用对特征做标准化之类的处理。

另外一个主要的优点是，基于树的方法是 “非参数” 的。这意味着它们不需要学习一组特定的参数。

1.1 银行贷款意向分析决策树示意图



例如，某客户的信息为：{职业、年龄，收入，学历}={工人、39，1800，小学}，你能判断出他是否有贷款意向？

1.2 构建一颗决策树

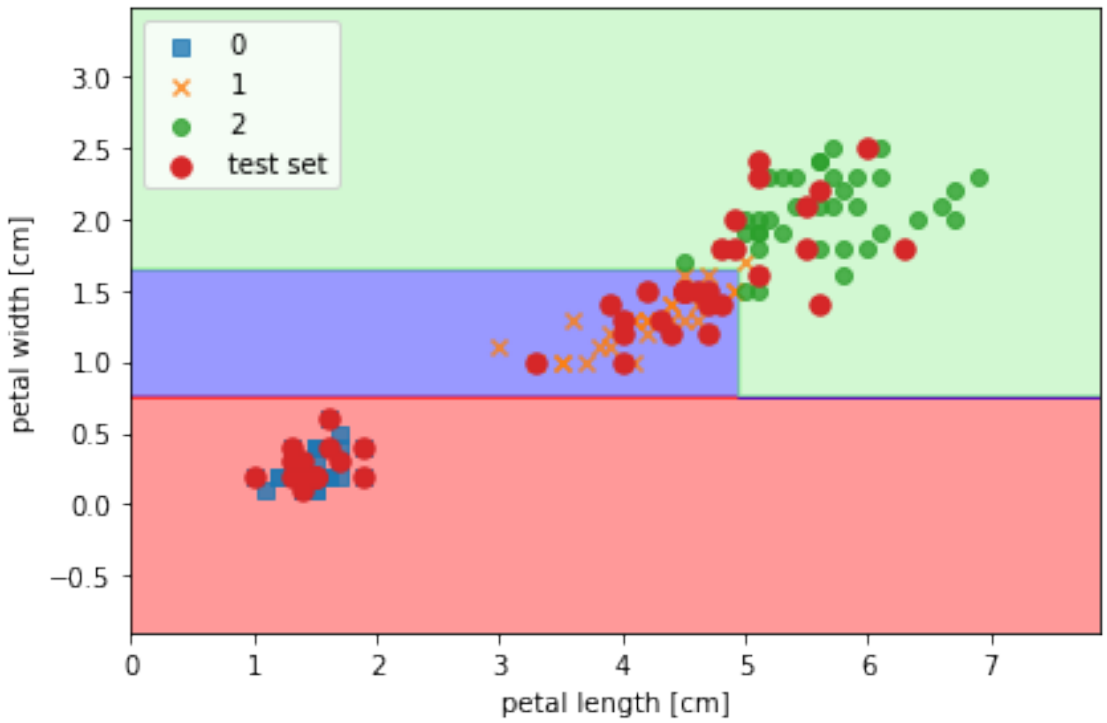
[1]:
```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=0)

# sc = StandardScaler()
# sc.fit(X_train)
```

```
# X_train_std = sc.transform(X_train) # standardize by mean & std
# X_test_std = sc.transform(X_test)
```

[2]:
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
%matplotlib inline

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # 画出所有样本
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8,
                    #c=cmap(idx),
                    marker=markers[idx], label=cl)

    # 画出测试样本
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
```

```
                #c='',
                alpha=1.0, linewidth=1, marker='o',
                s=55, label='test set')
```

[3]:
```
from sklearn.tree import DecisionTreeClassifier
# 调用 sklearn 的决策树方法，max_depth 为 3，不纯度量为 entropy
tree = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
tree.fit(X_train, y_train)

X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined,
                      classifier=tree, test_idx=range(105,150))

plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
# plt.savefig('./figures/decision_tree_decision.png', dpi=300)
```



1.3 可视化一颗决策树

scikit-learn 中决策树的可视化一般需要安装 graphviz

主要包括 graphviz 的安装和 python 的 graphviz 插件的安装。

1. 安装 graphviz，http://www.graphviz.org

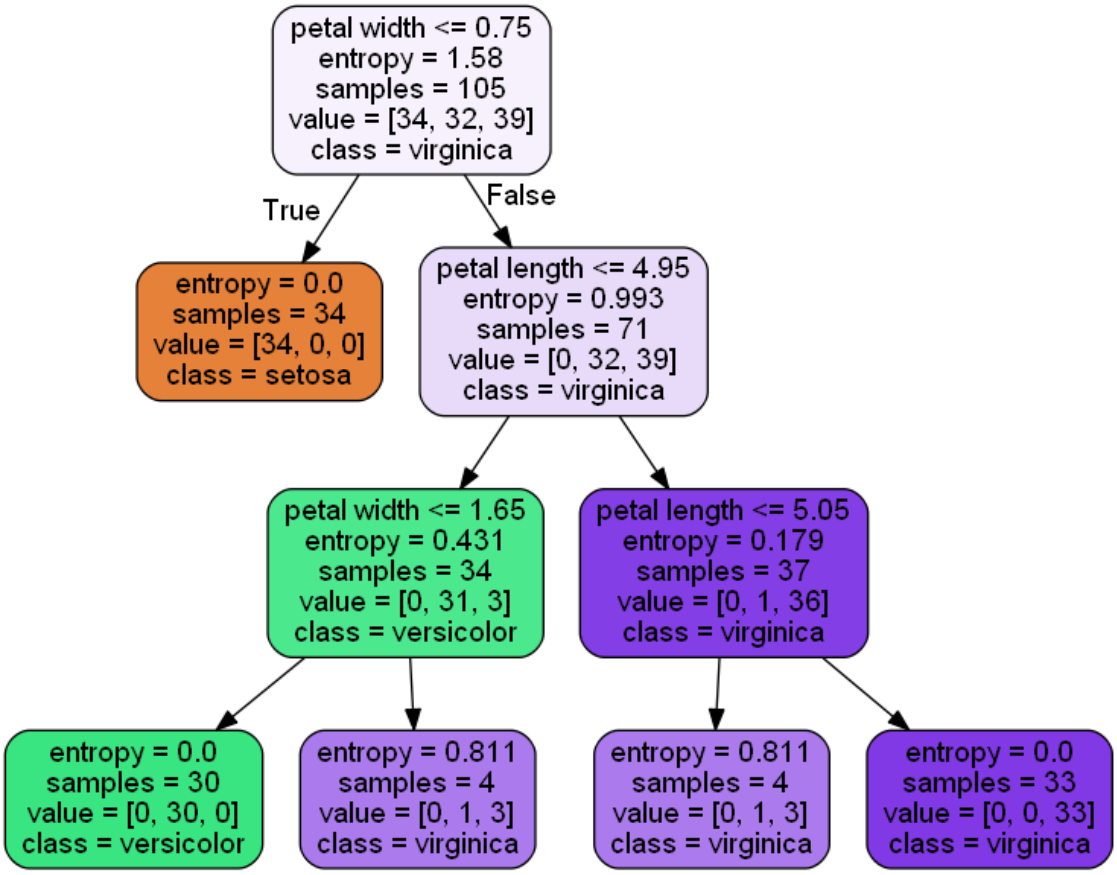
- linux，apt-get 或者 yum 的方法安装
- windows，官网下载 msi 文件安装

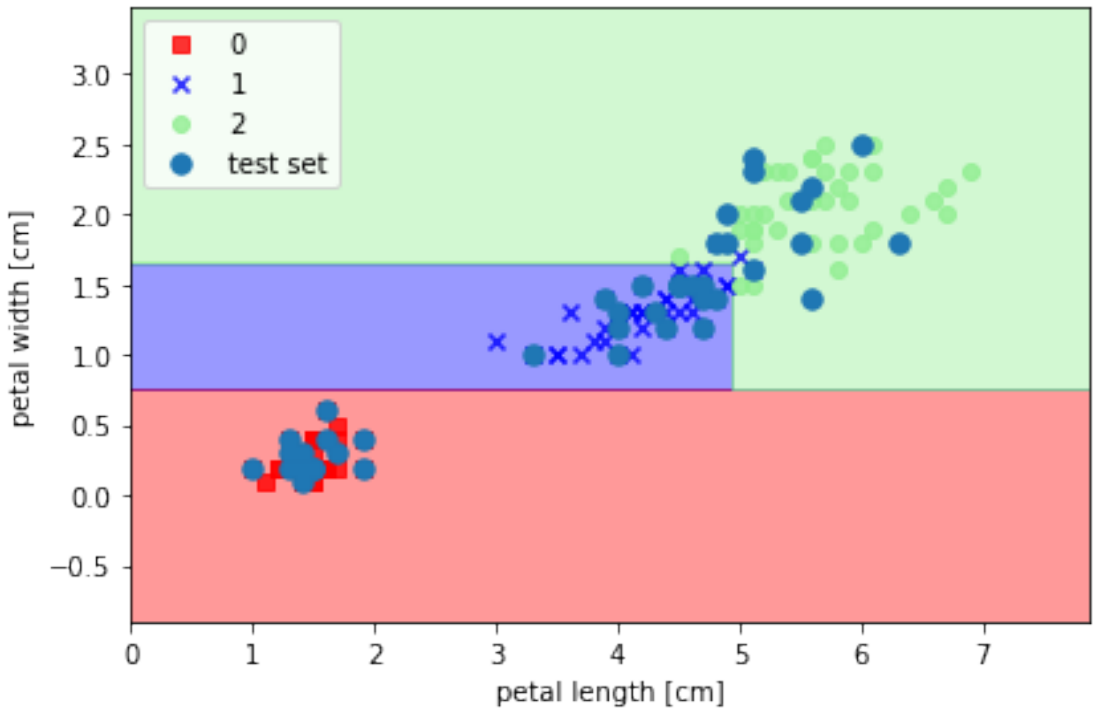
https://graphviz.gitlab.io/_pages/Download/Download_windows.html - 设置环境变量，将 graphviz 的 bin 目录加到 PATH

2. 安装 python 插件 graphviz：pip install graphviz

3. 安装 python 插件 pydotplus：pip install pydotplus

[4]:
```
from  sklearn.tree import DecisionTreeClassifier
import pydotplus
from sklearn.tree import export_graphviz
from IPython.display import Image
from IPython.display import display

dot_data = export_graphviz(
    tree,
    out_file=None,
    # the parameters below are new in sklearn 0.18
    feature_names=['petal length', 'petal width'],
    class_names=['setosa', 'versicolor', 'virginica'],
    filled=True,
    rounded=True)

graph = pydotplus.graph_from_dot_data(dot_data)
display(Image(graph.create_png()))
```


petal width <= 0.75
entropy = 1.58
samples = 105
value = [34, 32, 39]
class = virginica
True
False
entropy = 0.0
samples = 34
value = [34, 0, 0]
class = setosa
petal length <= 4.95
entropy = 0.993
samples = 71
value = [0, 32, 39]
class = virginica
petal width <= 1.65
entropy = 0.431
samples = 34
value = [0, 31, 3]
class = versicolor
petal length <= 5.05
entropy = 0.179
samples = 37
value = [0, 1, 36]
class = virginica
entropy = 0.0
samples = 30
value = [0, 30, 0]
class = versicolor
entropy = 0.811
samples = 4
value = [0, 1, 3]
class = virginica
entropy = 0.811
samples = 4
value = [0, 1, 3]
class = virginica
entropy = 0.0
samples = 33
value = [0, 0, 33]
class = virginica



1.4 决策树的构建

决策树的构建是数据逐步分裂的过程，构建的步骤如下:

步骤 1：将所有的数据看成是一个节点，进入步骤 2；

步骤 2：从所有的数据特征中挑选一个数据特征对节点进行分割，进入步骤 3；

步骤 3：生成若干孩子节点，对每一个子节点进行判断，如果满足停止分裂的条件，进入步骤 4；否则，进入步骤 2；

步骤 4：设置该节点是子节点，其输出的结果为该节点数量占比最大的类别。

从上述步骤可以看出，决策生成过程中有两个重要的问题：

（1）数据如何分割

（2）如何选择分裂的属性

（3）什么时候停止分裂

1.5 不同的纯度标准–分裂属性的选择

1.5.1 熵与基尼指数

设 X 是一个取有限个值的离散随机变量，其概率分布为：

$$P(X = x_i) = p_i$$

在信息论和概率统计中，熵 (entropy) 是表示随机变量不确定性的度量，定义为

$$entropy = -\sum_i p_i \log(p_i)$$

熵描述了数据的混乱程度，熵越大，混乱程度越高，也就是纯度越低；反之，熵越小，混乱程度越低，纯度越高。

基尼指数定义为

$$Gini(p) = \sum_{i=1}^{K} p_i(1 - p_i) = 1 - \sum_{i=1}^{K} p_i^2$$

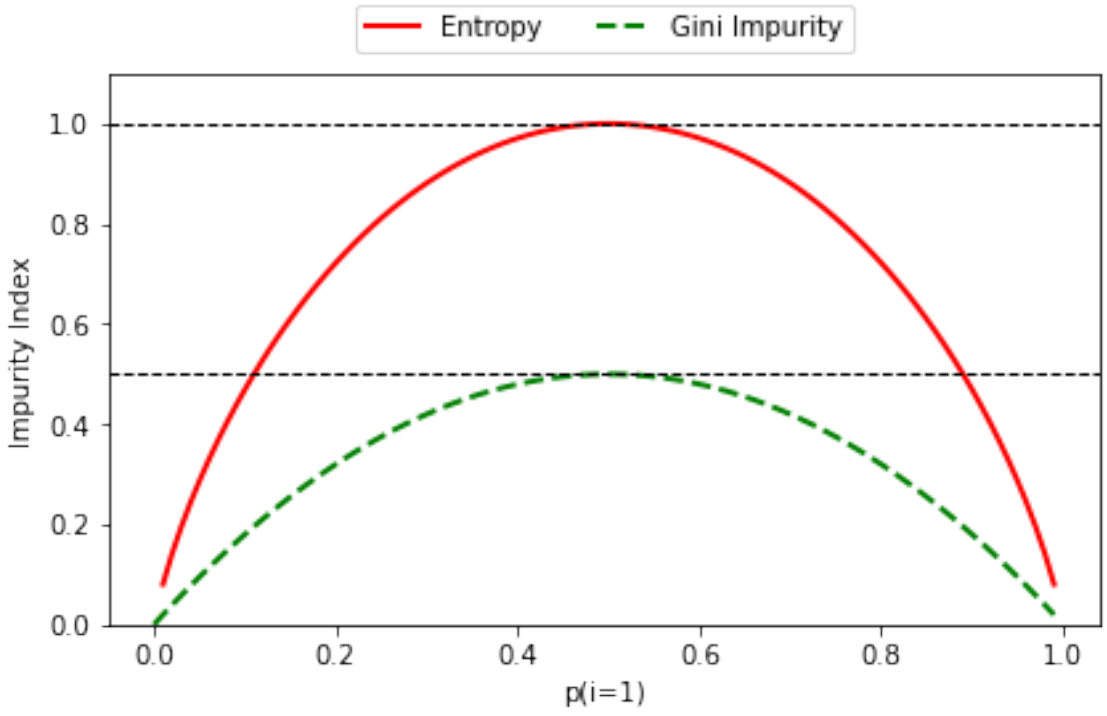
基尼指数性质是：1. 类别个数越少，基尼系数越低，2. 类别个数相同时，类别集中度越高，基尼系数越低。

当类别越少，类别集中度越高的时候，基尼系数越低；当类别越多，类别集中度越低的时候，基尼系数越高。

不同的算法实现对特征选择的准则是不一样的（不纯度）- 信息增益（ID3 算法）- 信息增益比（C4.5 算法）- 基尼指数（CART 算法）

[5]:
```
def gini(p):
    return p*(1 - p) + (1-p)*(1 - (1-p))

def entropy(p):
    return - p*np.log2(p) - (1 - p)*np.log2((1 - p))

x = np.arange(0.0, 1.0, 0.01)

ent = [entropy(p) if p != 0 else None for p in x]
```

```
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, gini(x)],
                  ['Entropy', 'Gini Impurity'],
                  ['-', '--'],
                  ['red', 'green']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)


# 画图
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
          ncol=3, fancybox=True, shadow=False)

ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('Impurity Index')
plt.tight_layout()
# plt.savefig('./figures/impurity.png', dpi=300, bbox_inches='tight')
```



[6]:
```
p = 0.25
entropy(p)
```

[6]: 0.8112781244591328

[7]:
```
p = 0.5
entropy(p)
```

[7]: 1.0

1.5.2 条件熵

设有随机变量 (X, Y)，条件熵 $H(Y|X)$ 表示在已知随机变量 X 的条件下随机变量 Y 的不确定性。随机变量 X 给定的条件下随机变量 Y 的条件熵 $H(Y|X)$ 定义为 X 给定条件下 Y 的条件概率分布的熵对 X 的数学期望

$$H(Y|X) = \sum_{i=1}^{n} p_i H(Y|X = x_i)$$

其中 $p_i = P(X = x_i), \quad i = 1, 2, ..., n.$

1.5.3 信息增益

信息增益表示得知特征 X 的信息而使得类的信息的不确定性减少程度。特征 A 对训练数据集 D 的信息增益 $g(D, A)$ 为集合 D 的熵 $H(D)$ 与特征 A 给定条件下 D 的条件熵 $H(D|A)$ 之差，即

$$g(D, A) = H(D) - H(D|A).$$

信息增益的算法:

输入：训练数据集 D，$|D|$ 为样本容量, 有 K 个类 $C_k, k = 1, 2, ..., K$, $|C_k|$ 为属于类 C_k 的样本个数。特征 A 有 n 个不同取值，根据特征 A 的取值将 D 划分为 n 个子集 $D_1, D_2, ..., D_n, D_{ik}$ 为子集 D_i 中属于类 C_k 的集合。

输出：特征 A 对训练数据集 D 的信息增益 $g(D, A)$

1. 计算数据集 D 的熵 $H(D)$

$$H(D) = -\sum_{k=1}^{K} \frac{|C_k|}{|D|} \log \frac{|C_k|}{|D|}$$

2. 计算特征 A 对训练数据集 D 的条件熵 $H(D|A)$

$$H(D|A)=\sum_{i=1}^{n}\frac{|D_i|}{|D|}H(D_i)=-\sum_{i=1}^{n}\frac{|D_i|}{|D|}\sum_{k=1}^{K}\frac{|D_{ik}|}{|D_i|}\log\frac{|D_{ik}|}{|D_i|}.$$

3. 计算信息增益

$$g(D,A)=H(D)-H(D|A)$$

1.5.4 例子

Day	Temperatrue	Outlook	Humidity	Windy	PlayGolf?
07-05	hot	rainy	high	false	no
07-06	hot	sunny	high	true	no
07-07	hot	overcast	high	false	yes
07-09	cool	rainy	normal	false	yes
07-10	cool	overcast	normal	true	yes
07-12	mild	sunny	high	false	no
07-14	cool	sunny	normal	false	yes
07-15	mild	rainy	normal	false	yes
07-20	mild	sunny	normal	true	yes
07-21	mild	overcast	high	true	yes
07-22	hot	overcast	normal	false	yes
07-23	mild	rainy	high	true	no
07-26	cool	sunny	normal	true	no
07-30	mild	rainy	high	false	yes

计算整个数据集的熵

$$p_1=\frac{Count(no)}{Count(no)+Count(yes)}=\frac{5}{14}$$

$$p_2=\frac{Count(yes)}{Count(no)+Count(yes)}=\frac{9}{14}$$

$$H(D)=-\frac{5}{14}\log\frac{5}{14}-\frac{9}{14}\log\frac{9}{14}=0.9403$$

假如把 Outlook 作为分裂特征

$$H(Outlook=sunny)=-\frac{2}{5}\log\frac{2}{5}-\frac{3}{5}\log\frac{3}{5}=0.971$$

$$H(Outlook = overcast) = -1\log 1 - 0\log 0 = 0$$
$$H(Outlook = rainy) = -\frac{3}{5}\log\frac{3}{5} - \frac{2}{5}\log\frac{2}{5} = 0.693$$

$$H(D|A) = -\frac{5}{14}\cdot 0.971 - \frac{4}{14}\cdot 0 + \frac{5}{14}\cdot 0.971 = 0.693$$

信息增益为

$$g(D, Outlook) = 0.9403 - 0.693 = 0.2473$$

假如用 Day 来做为特征，$(|) = 0$，那么 $(,) = 0.9403$，显然，每一天都可以将样本分开，这种样本分隔的结果就是形成了一棵叶子数量为 14，深度只有两层的树。显然这种特征对于样本的分隔没有任何意义。解决办法是对树分支过多的情况进行惩罚，这样就引入了信息增益比。

1.5.5 信息增益比

特征 A 对训练数据集 D 的信息增益比 $g_R(D, A)$ 定义为其信息增益 $g(D, A)$ 与训练数据集 D 关于特征 A 的值的熵 $H_A(D)$ 之比，即

$$g_R(D, A) = \frac{g(D, A)}{H_A(D)}$$

其中，$H_A(D) = -\sum_{i=1}^{n}\frac{|D_i|}{|D|}log\frac{|D_i|}{|D|}$, n 是特征 A 取值的个数。

1.5.6 停止分裂的条件

（1）最小节点数据量

当节点的数据量小于一个指定的数量时，不继续分裂。两个原因：一是数据量较少时，再做分裂容易强化噪声数据的作用；二是降低树生长的复杂性。提前结束分裂一定程度上有利于降低过拟合的影响。

（2）熵或者基尼值小于阀值。

由上述可知，熵和基尼值的大小表示数据的复杂程度，当熵或者基尼值过小时，表示数据的纯度比较大，如果熵或者基尼值小于一定程度数，节点停止分裂。

（3）决策树的深度达到指定的条件

节点的深度可以理解为节点与决策树跟节点的距离，如根节点的子节点的深度为 1，因为这些节点与跟节点的距离为 1，子节点的深度要比父节点的深度大 1。决策树的深度是所有叶子节点的最大深度，当深度到达指定的上限大小时，停止分裂。

（4）所有特征已经使用完毕，不能继续进行分裂。

被动式停止分裂的条件，当已经没有可分的属性时，直接将当前节点设置为叶子节点。

1.6 使用 Python 实现二分决策树

[8]:
```
from collections import Counter
import numpy as np


# 构建一个类，来表征二分树的结构
# Tree 里的属性除了包括左右节点的 Tree 之外，还有此节点中包括数据的标签及其熵值，
# 然后还有要切分 feature 的 idex
class Tree:
    """ Binary Tree"""
    def __init__(self, labels, split_idx=None,children_left=None,
                 children_right=None):
        self.children_left = children_left
        self.children_right = children_right
        self.labels = labels
        self.split_idx = split_idx
        self.entropy = calc_entropy(self.labels)

    def predict(self):
        # 找到最高频的元素，“投票”
        most_freq = np.bincount(self.labels).argmax()
        return most_freq

    def __repr__(self, level=0):
        """ make it easy to visualize a tree"""
        prefix = "\t" * level
        string = prefix + "entropy = {}, labels = {}, [0 的个数, 1 的个数] =␣
↪{}\n".format(
            self.entropy, self.labels, np.bincount(self.labels, minlength=2))
        if self.split_idx is not None:
            string += prefix + "split on Column {}\n".format(self.split_idx)
            string += prefix + "True:\n"
            string += self.children_left.__repr__(level+1)
            string += prefix + "False:\n"
```

```
            string += self.children_right.__repr__(level+1)
        return string


# 计算一组数据里的熵值
def calc_entropy(labels):
    """ calculate entropy from an array of labels"""
    size = float(len(labels))
    cnt = Counter(labels)
    entropy = 0
    for label in set(labels):
        prob = cnt[label] / size
        entropy += -1 * prob * np.log2(prob)
    return entropy


# 不同的决策树算法（如 ID3, C4.5, CART 等）会用不同的标准来选择要切分的 feature
# 这里使用的是信息增益，即 feature 切分前后的熵值变化
def choose_best_feature_to_split(features, labels):
    """ 选择信息增益最大的特征作为最优分裂点"""
    num_features = features.shape[1]
    base_entropy = calc_entropy(labels)

    best_info_gain = 0
    best_feature = None

    for i in range(num_features):
        new_entropy = 0
        for value in [0, 1]:# 每个 feature 只有 0,1 两个取值
            new_labels = labels[features[:, i] == value]
            weight = float(len(new_labels)) / len(labels)
            new_entropy += weight * calc_entropy(new_labels)
        info_gain = base_entropy - new_entropy
        if info_gain > best_info_gain:
            best_info_gain = info_gain
            best_feature = i
```

```
    return best_feature


def create_decision_tree(features, labels,current_depth=0, max_depth=10):
    """ 递归建树"""
    tree = Tree(labels)

    # 定义停止条件，当在某节点上的所有数据都是一类，就停止
    if len(set(labels)) == 1:
        return tree
    # 当最大深度达到，也停止
    if current_depth >= max_depth:
        return tree

    # 在找到的最优特征上分裂
    best_feature = choose_best_feature_to_split(features, labels)
    if best_feature is None:
        return tree

    # 递归构建子树
    msk = (features[:, best_feature] == 1)
    tree.split_idx = best_feature
    tree.children_left = create_decision_tree(
        features[msk], labels[msk], current_depth+1)
    tree.children_right = create_decision_tree(
        features[~msk], labels[~msk], current_depth+1)
    return tree
```

[9]:
```
# 模拟一组数据来测试
data = np.array([[1, 0],
                 [1, 1],
                 [0, 1],
                 [0, 0]])
labels = np.array([1, 0, 0, 0])
```

```
tree = create_decision_tree(data, labels)
```

[10]:
```
tree
```

[10]:
```
entropy = 0.8112781244591328, labels = [1 0 0 0], [0 的个数, 1 的个数] = [3 1]
split on Column 0
True:
        entropy = 1.0, labels = [1 0], [0 的个数, 1 的个数] = [1 1]
        split on Column 1
        True:
                entropy = 0.0, labels = [0], [0 的个数, 1 的个数] = [1 0]
        False:
                entropy = 0.0, labels = [1], [0 的个数, 1 的个数] = [0 1]
False:
        entropy = 0.0, labels = [0 0], [0 的个数, 1 的个数] = [2 0]
```

[11]:
```
# 遍历二分树，来得到分类预测
def _classify(tree, data_row):
    """ 对新数据进行预测"""
    if tree.split_idx is None: # 是不是叶子节点，是的话，直接 predict
        return tree.predict()

    split_idx = tree.split_idx
    if data_row[split_idx]:  # 分裂条件是否为 True，满足分裂条件，往左；
                             # 不然，往右（比如颜色是 1，往左，不然，往右）
        return _classify(tree.children_left, data_row)
    else:
        return _classify(tree.children_right, data_row)

def classify(tree, data):
    data = np.array(data)
    num_row = data.shape[0]
    results = np.empty(shape=num_row)
    for i in range(num_row):
        results[i] = _classify(tree, data[i, :])
    return results
```

[12]:
```
new_data = [[0, 0],
            [1, 0]]

classify(tree, new_data)
```

[12]: array([0., 1.])

2 回归树

2.1 原理

回归树是可以用于回归的决策树模型，一个回归树对应着输入空间（即特征空间）的一个划分以及在划分单元上的输出值。与分类树不同的是，回归树对输入空间的划分采用一种启发式的方法，会遍历所有输入变量，找到最优的切分变量 j 和最优的切分点 s，即选择第 j 个特征 x_j 和它的取值 s 将输入空间划分为两部分，然后重复这个操作。

而如何找到最优的 j 和 s 是通过比较不同的划分的误差来得到的。一个输入空间的划分的误差是用真实值和划分区域的预测值的平方和来衡量的，即

$$\sum_{x^{(i)} \in R_m} (y^{(i)} - f(x^{(i)}))^2$$

其中，$f(x_i)$ 是每个划分单元的预测值，这个预测值是该单元内每个样本点的值的均值，即

$$f(x^{(i)}) = c_m = ave(y^{(i)} | x^{(i)} \in R_m), \quad m = 1, 2$$

其中，$R_1(j, s) = \{x | x_j \leq s\}$ 和 $R_2(j, s) = \{x | x_j > s\}$ 是被划分后的两个区域。

CART 选择切分变量 j 和切分点 s 的公式如下：

$$\min_{j,s} \left(\min_{c_1} \sum_{x^{(i)} \in R_1(j,s)} (y^{(i)} - c_1)^2 + \min_{c_2} \sum_{x^{(i)} \in R_2(j,s)} (y^{(i)} - c_2)^2 \right)$$

2.2 举例

举一个简单实例。训练数据见下表，目标是得到一棵最小二乘回归树。

x	1	2	3	4	5	6	7	8	9	10
y	5.56	5.7	5.91	6.4	6.8	7.05	8.9	8.7	9	9.05

选择最优切分变量 j 与最优切分点 s, 在本数据集中，只有一个变量，因此最优切分变量自然是 x。

接下来我们考虑 9 个切分点 (1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5).

例如，取 $s=1.5$，此时 $R_1=\{1\}, R_2=\{2,3,4,5,6,7,8,9,10\}$，这两个区域的输出值分别为：

$$
\begin{aligned}
c_1 &= 5.56 \\
c_2 &= \frac{1}{9}(5.7+5.91+6.4+6.8+7.05+8.9+8.7+9+9.05) = 7.50
\end{aligned}
$$

得到下表：

s	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
c1	5.56	5.63	5.72	5.89	6.07	6.24	6.62	6.88	7.11
c2	7.5	7.73	7.99	8.25	8.54	8.91	8.92	9.03	9.05

把 c_1, c_2 的值代入误差计算公式，如：$m(1.5)=0+15.72=15.72$。同理，可获得下表：

s	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5
m(s)	15.72	12.07	8.36	5.78	3.91	1.93	8.01	11.73	15.74

显然取 $s=6.5$ 时，$m(s)$ 最小。因此，第一个划分变量 $j=x, s=6.5$.

对两个子区域继续调用上述步骤，假设在生成 3 个区域之后停止划分，那么最终生成的回归树形式如下：

$$
T=\begin{cases}
5.72 & x \leq 3.5 \\
6.75 & 3.5 \leqslant x \leq 6.5 \\
8.91 & x > 6.5
\end{cases}
$$

diabetes 是一个关于糖尿病的数据集，该数据集包括 442 个病人的生理数据及一年以后的病情发展情况。

[13]:
```
import numpy as np
import matplotlib.pyplot as plt
import random

from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split

# 使用 scikit-learn 自带的一个糖尿病病人的数据集
diabetes = datasets.load_diabetes()

print(diabetes.keys())
diabetes['feature_names']
```

```
dict_keys(['data', 'target', 'DESCR', 'feature_names', 'data_filename',
'target_filename'])
```

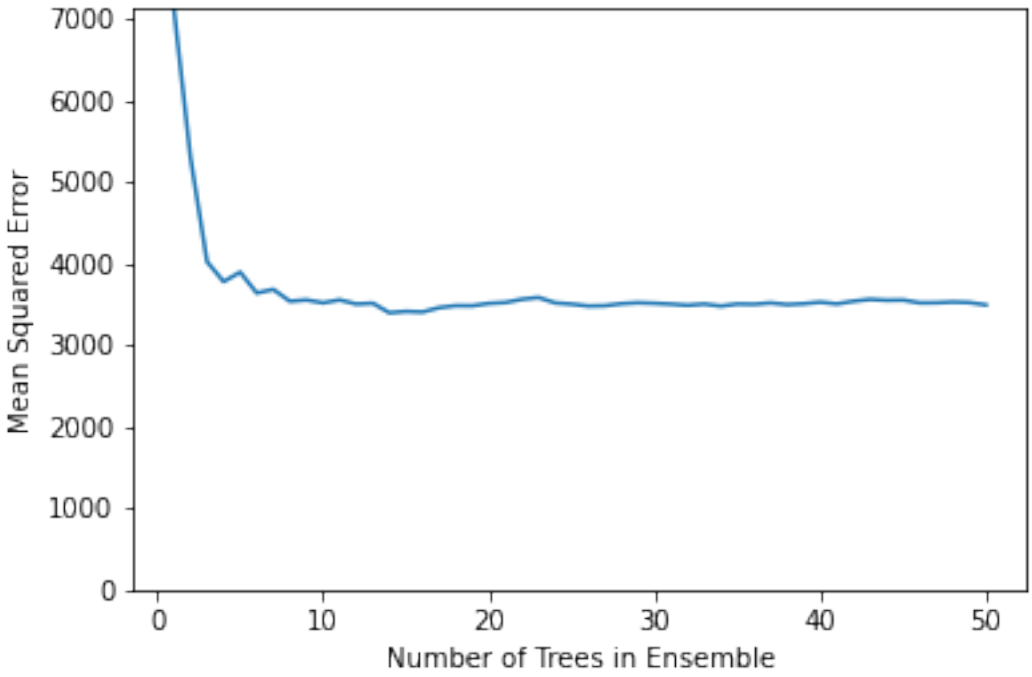
[13]:
```
['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

随机选择部分训练数据和部分特征训练模型

[14]:
```
# 拆分成训练集和测试集，测试集大小为原始数据集大小的 1/4
X_train,X_test,y_train,y_test = train_test_split(diabetes.data,diabetes.
 ↪target,test_size=0.25,random_state=0)

# 特征的个数
ncols = len(diabetes['feature_names'])
# train a series of models on random subsets of the training data
# collect the models in a list and check error of composite as list grows

# 生成树的最大个数
num_trees_max = 50

# 树的最大深度
tree_depth = 12

# 模型中使用的特征数
```

```
n_attr = int(0.6*ncols)

# 初始化保存数据的列表
mode_list = []
index_list = []
pred_list = []

for i_trees in range(num_trees_max):

    # 将模型加入列表
    mode_list.append(DecisionTreeRegressor(max_depth=tree_depth))

    # 随机地选择特征
    idx_attr = random.sample(range(ncols), n_attr)
    idx_attr.sort()
    index_list.append(idx_attr)

    # 随机选择训练数据
    idx_rows = []
    for i in range(int(0.8 * len(X_train))):
        idx_rows.append(random.choice(range(len(X_train))))
    idx_rows.sort()

    # 构造训练数据
    x_rf_train = []
    y_rf_train = []

    for i in range(len(idx_rows)):
        temp = [X_train[idx_rows[i]][j] for j in idx_attr]
        x_rf_train.append(temp)
        y_rf_train.append(y_train[idx_rows[i]])

    # 训练模型
    mode_list[-1].fit(x_rf_train, y_rf_train)

    # 选择与训练数据相同的特征构造测试数据
```

```
    x_rf_test = []
    for xx in X_test:
        temp = [xx[i] for i in idx_attr]
        x_rf_test.append(temp)

    latest_out_sample_prediction = mode_list[-1].predict(x_rf_test)
    pred_list.append(list(latest_out_sample_prediction))

pred_array = np.array(pred_list)
y_test = y_test.reshape(1,-1)
all_predictions = []
for i_models in range(num_trees_max):
    # 多个模型预测值的平均值作为预测结果
    all_predictions.append(np.mean(pred_array[:i_models+1,:],axis=0))

mse = np.sum((np.array(all_predictions)-y_test)**2, axis=1)/len(X_test)
n_models = [i + 1 for i in range(len(mode_list))]

plt.plot(n_models, mse)
plt.axis('tight')
plt.xlabel('Number of Trees in Ensemble')
plt.ylabel('Mean Squared Error')
plt.ylim((0.0, max(mse)))
plt.show()

print('Minimum MSE: ',min(mse))
```

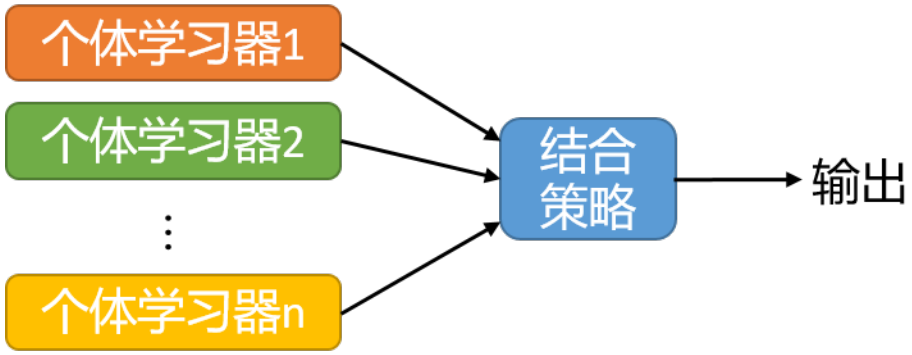



```
Minimum MSE:  3396.2576777630043
```

随着树的增多误差是越来越小的，最后达到稳定值。

3 集成学习

对于训练集数据，我们通过训练若干个个体学习器，通过一定的结合策略，就可以最终形成一个强学习器，以达到博采众长的目的。



集成学习有两个主要的问题需要解决，第一是如何得到若干个个体学习器，第二是如何选择一种结

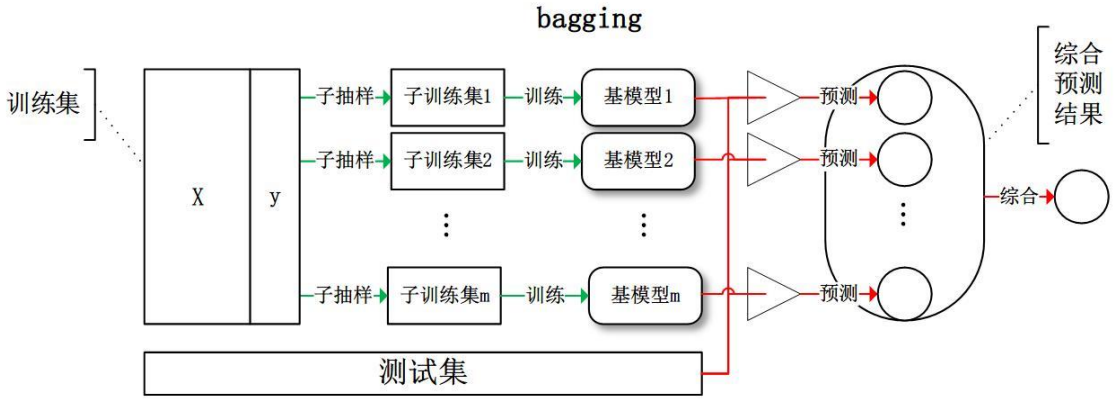
合策略，将这些个体学习器集合成一个强学习器。

根据个体学习器是否存在依赖关系分为两类：- 个体学习器之间存在强依赖关系，一系列个体学习器基本都需要串行生成，代表算法是 boosting 系列算法，- 个体学习器之间不存在强依赖关系，一系列个体学习器可以并行生成，代表算法是 bagging 和随机森林系列算法。

3.1 集成学习之 bagging

3.1.1 原理

bagging 的个体弱学习器的训练集是通过随机采样得到的。通过 T 次的随机采样，我们就可以得到 T 个采样集，对于这 T 个采样集，我们可以分别独立的训练出 T 个弱学习器，再对这 T 个弱学习器通过结合策略来得到最终的强学习器。



3.1.2 随机森林

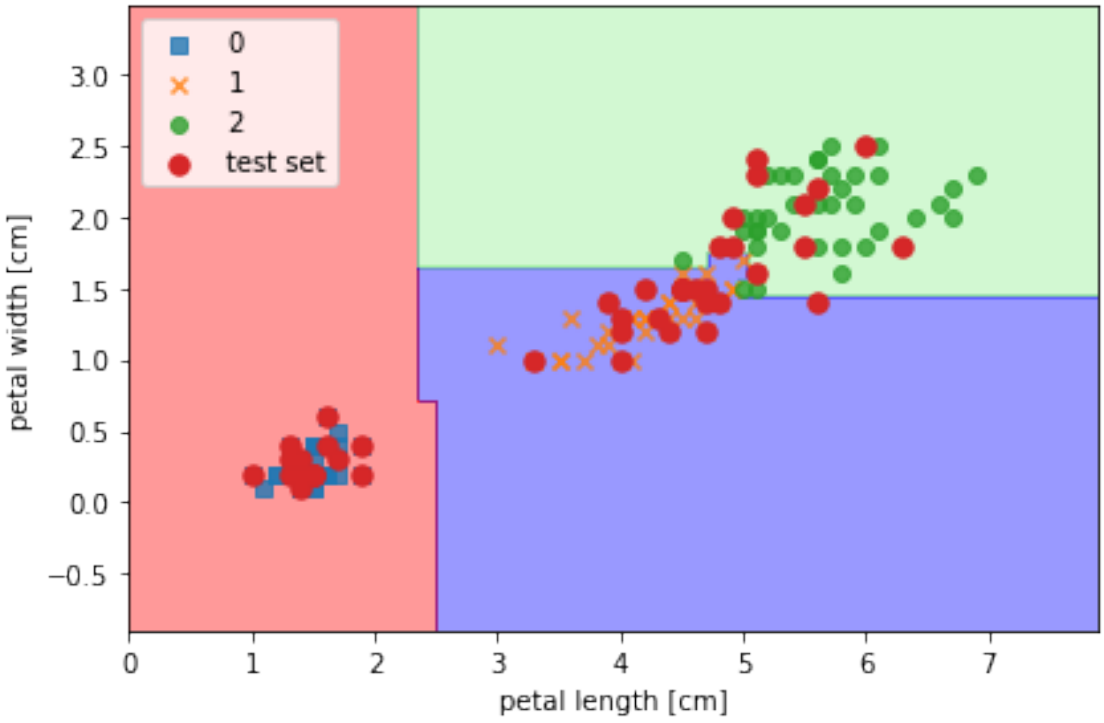
随机森林就是通过集成学习的思想将多棵树集成的一种算法，它的基本单元是决策树。

随机森林生成过程：

1. 随机且有放回地从训练集中抽取 N 个训练样本（这种采样方式称为自助采样法（bootstrap sample）方法）；
2. 如果每个样本的特征维度为 M，指定一个常数 m<<M，随机地从 M 个特征中选取 m 个特征子集，每次树进行分裂时，从这 m 个特征中选择最优的；
3. 重复步骤 1 和步骤 2，k 次；
4. 最后的预测是每棵树的判断做投票。

[15]:
```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

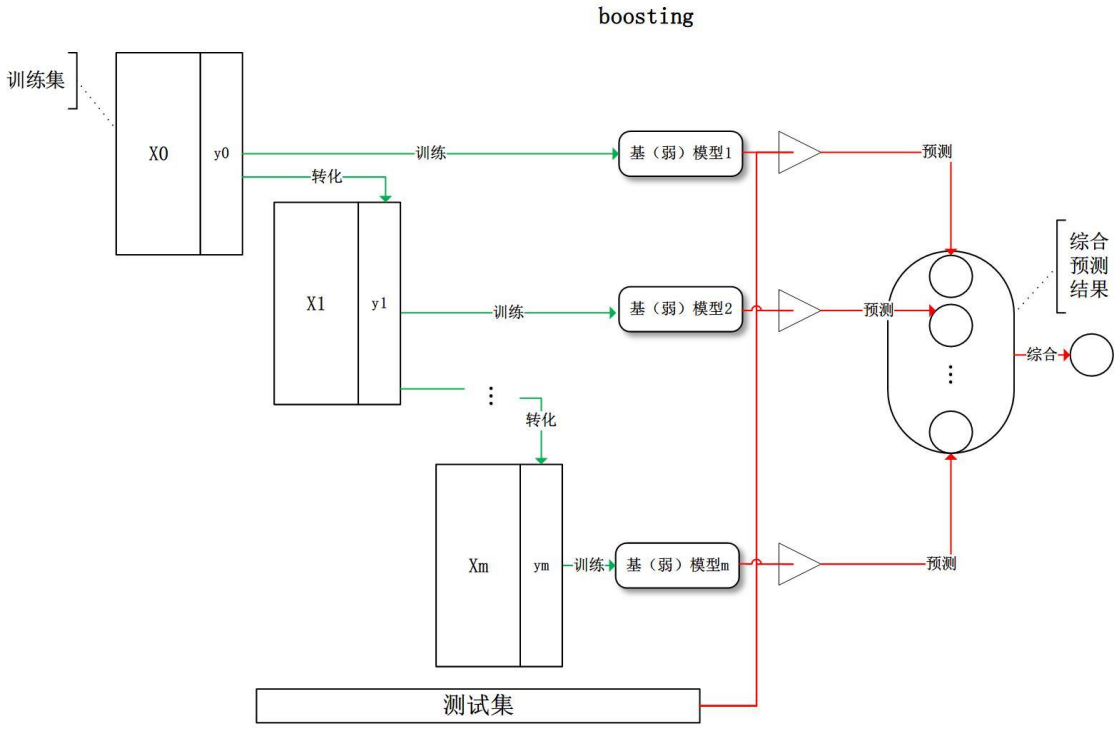
```
from sklearn.ensemble import RandomForestClassifier

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(
         X, y, test_size=0.3, random_state=0)

X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))

# 10 decision trees
forest = RandomForestClassifier(criterion='entropy',
                                n_estimators=10,
                                random_state=1)

forest.fit(X_train, y_train)

plot_decision_regions(X_combined, y_combined, classifier=forest,␣
 ↪test_idx=range(105,150))

plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
# plt.savefig('./figures/random_forest.png', dpi=300)
```



3.2 集成学习之 boosting

3.2.1 原理

Boosting 算法的工作机制是首先从训练集用初始权重训练出一个弱学习器 1，根据弱学习的学习误差率表现来更新训练样本的权重，使得之前弱学习器 1 学习误差率高的训练样本点的权重变高，使得这些误差率高的点在后面的弱学习器 2 中得到更多的重视。然后基于调整权重后的训练集来训练弱学习器 2.，如此重复进行，直到弱学习器数达到事先指定的数目 T，最终将这 T 个弱学习器通过结合策略进行整合，得到最终的强学习器。



3.2.2 Adaboost

AdaBoost 中 Ada 为 Adaptive 的简写, 适应性（adaptive）是指后续的弱学习器为了支持先前分类器分类错误的实例而进行调整。

AdaBoost 算法的每一次迭代都会为训练样本赋予新的权重。最开始的训练样本权重相同，均为 $1/N$，所以第一次迭代相当于在数据原有的样子上训练分类器，而在接下来的迭代中，训练集中的每一个样本权重都可能被修改，而分类算法会在已修改权重的训练样本上训练下一个分类器。那些在上一次迭代中被分类器分类错误的样本的权重会被增加，而分类正确的样本权重会减小，这样随着迭代的进行，难以正确分类的样本会被赋予越来越大的权重，迫使接下来的分类器越来越关注这些在之前错误分类的样本.

算法 输入：训练数据集 $\{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$, 其中 $x_i \in R^n$, $y_i \in \{-1, +1\}$.

输出：最终分类器 $G(x)$ 1. 初始化训练数据权重 $w_{1,i} = 1/N, i = 1, 2, \cdots, N$.

2. 对 $m = 1, 2 \cdots, M$

- (a) 使用权值 $w_{m,i}$ 训练分类器 $G_m(x)$
- (b) 计算 $G_m(x)$ 在训练数据集上的分类误差率

$$err_m = \sum_{i=1}^{N} w_{m,i} I(y_i \neq G_m(x_i))$$

- (c) 计算 $G_m(x)$ 的系数

$$\alpha_m = \frac{1}{2}\log\frac{1-err_m}{err_m}$$

- (d) 更新权值

$$w_{m+1,i} = \frac{w_{m,i}\cdot\exp(-\alpha_m y_i G_m(x_i))}{Z_m}, \quad i = 1, 2, \cdots, N$$

其中 Z_m 是规范化因子

$$Z_m = \sum_{i=1}^{N} w_{m,i}\cdot\exp(-\alpha_m y_i G_m(x_i))$$

3. 构建基本分类器的线性组合

$$f(x) = \sum_{i=1}^{M}\alpha_m G_m(x)$$

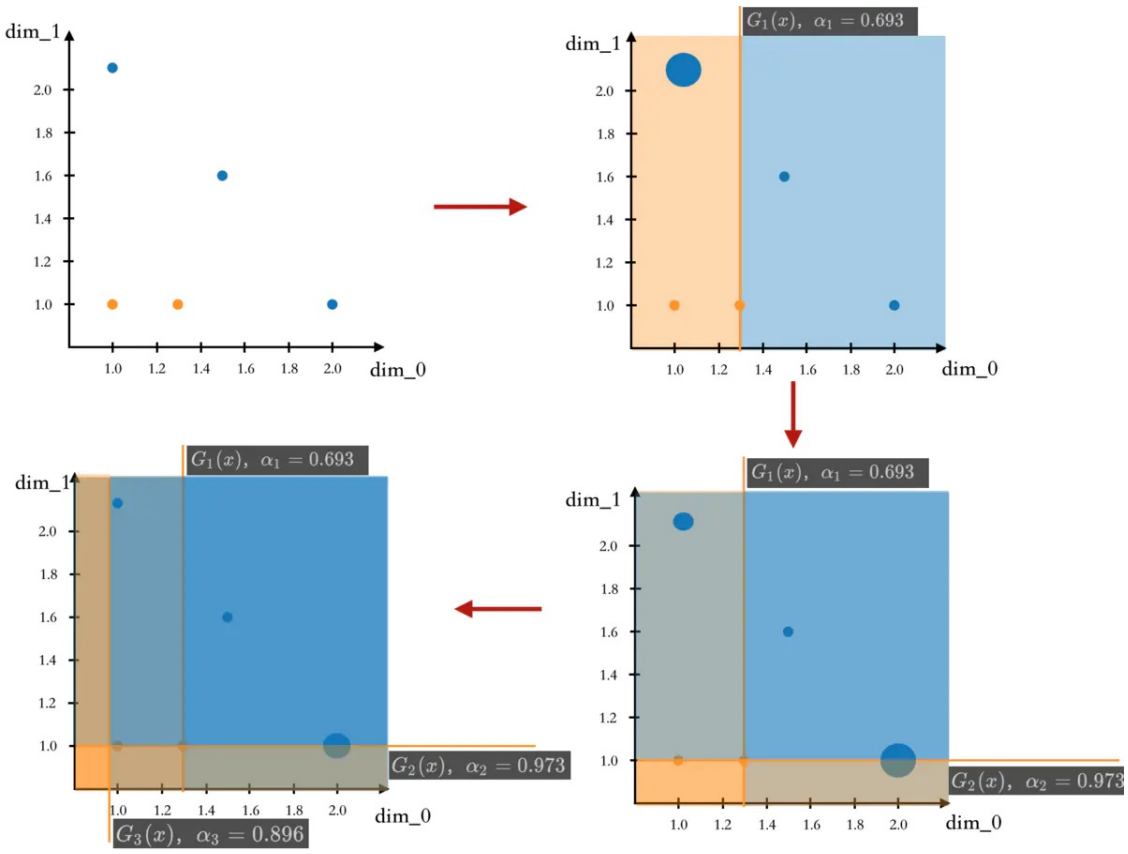
得到最终分类器

$$G(x) = sign(f(x)) = sign(\sum_{i=1}^{M}\alpha_m G_m(x))$$

算法在 2(a) 行对加权训练样本训练分类器 $G_m(x)$；2(b) 行计算加权误差率；2(c) 行计算分类器 $G_m(x)$ 的权重 α_m；2(d) 行用 2(c) 行中计算的 α_m 更新每个样本的权重 $w_{m,i}$。用因子 $\exp(\alpha_m)$ 放大分类器 $G_m(x)$ 错误分类的样本权重，增加它们在训练下一个分类器 $G_{m+1}(x)$ 时的影响力。

举例 如下图左上所示，有两个特征 dim_0，dim_1，训练数据为 5 个点：{(1.0, 2.1)，(1.5, 1.6)，(1.3, 1.0)，(1.0, 1.0)，(2.0, 1.0)}，类别标签为 {1.0，1.0，-1.0，-1.0，1.0}（1.0 为蓝色点，-1.0 为橙色点）。

初始时，所有点权重一样（点大小表示权重大小）。选用的弱分类器是单层决策树（decision stump），它是一种简单的决策树，仅有两个结点，通过给定的阈值，进行分类。



首先，如上图右上所示，我们在 dim_0 选了一个相对较好的弱分类器 $G_1(x)$，dim_0 <= 1.3 为-1.0，即橙色，dim_0 > 1.3 为 1.0，即蓝色。此时，我们可以看到点 (1.0, 2.1) 被错误分类为橙色，因此增加权重，根据权重的计算方法，算出 $\alpha_1 = \log \frac{1-\frac{1}{5}}{\frac{1}{5}} = 0.693$。

其次，如上图右下所示，我们在 dim_1 选了一个弱分类器 $G_2(x)$，dim_1 <= 1.0 为-1.0，即橙色，dim_0 > 1.0 为 1.0，即蓝色，根据权重的计算方法，算出 $\alpha_2 = 0.973$。此时，我们可以看到点 [2.0, 1.0] 被错误分类为橙色，因此增加权重；

最后，如上图左下所示，我们在 dim_0 选了一个弱分类器 $G_3(x)$，能够将之前分错的两个点都能正确分类，dim_0 <= 0.9 为-1.0，即橙色，dim_0 > 0.9 为 1.0，即蓝色，根据权重的计算方法，算出 $\alpha_3 = 0.896$。

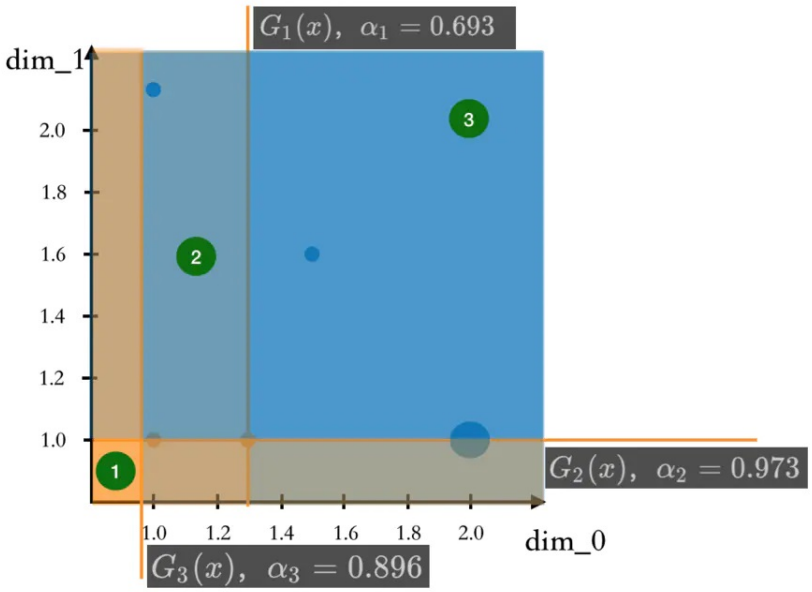
此时我们获得 3 个弱分类器：

$$G_1(x), \quad \alpha_1 = 0.693$$

$$G_2(x), \quad \alpha_1 = 0.973$$

$$G_3(x), \quad \alpha_1 = 0.896$$

那么使用上面三个弱分类器预测一下三个绿色的点所属类别吧，如下图所示：

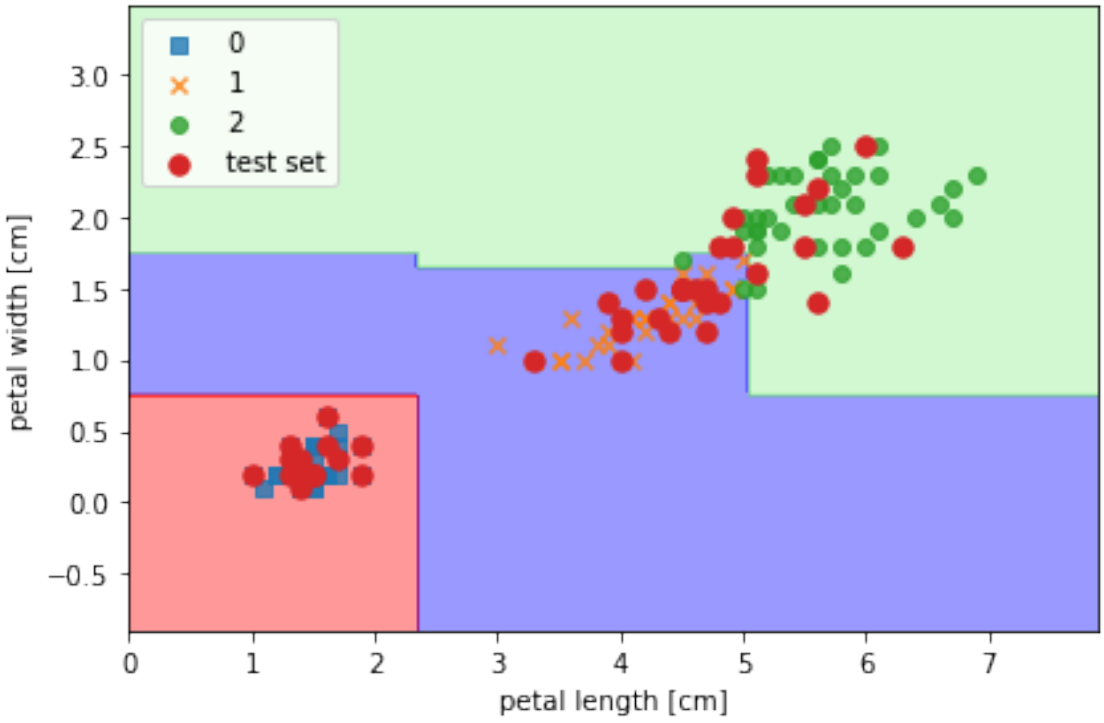


点 1：$(-0.693 + (-0.973) + (-0.896) = -2.562$ 为橙色点；

点 2：$(-0.693) + 0.973 + 0.896 = 1.176$ 为蓝色点；

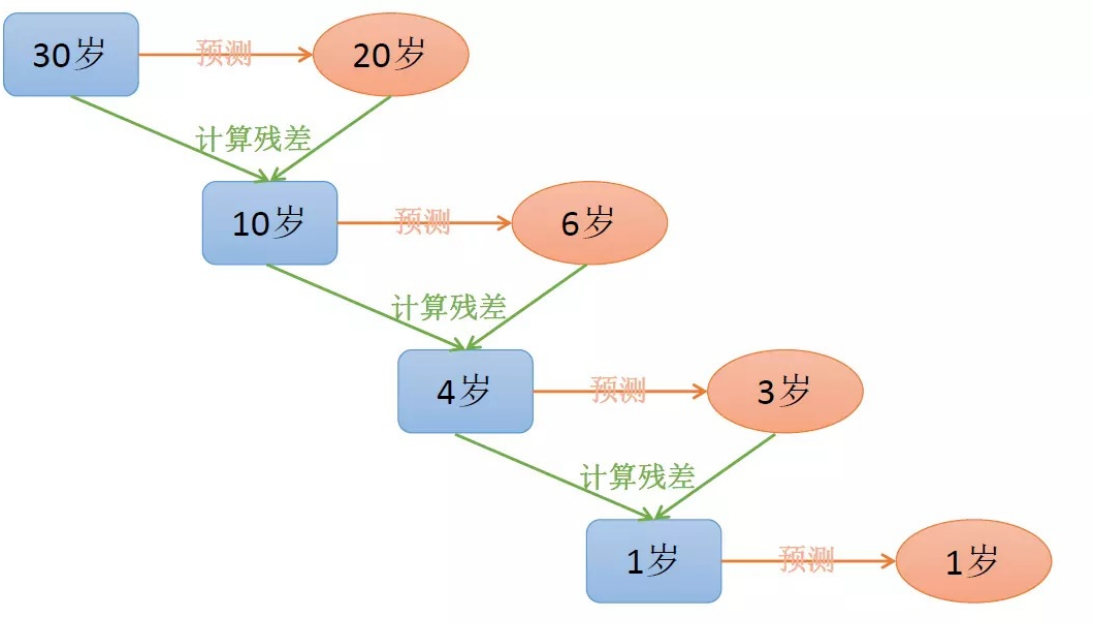
点 3：$0.693 + 0.973 + 0.896 = 2.562$ 为蓝色点；

[16]:
```
from sklearn.ensemble import AdaBoostClassifier
from  sklearn.tree import DecisionTreeClassifier
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,␣
 ↪min_samples_split=20, min_samples_leaf=5),
                         algorithm="SAMME",# 用对样本集分类效果作为弱学习器权重
                         n_estimators=200, # 最大的弱学习器的个数
                         learning_rate=0.8) # 每个弱学习器的权重缩减系数
bdt.fit(X_train, y_train)

plot_decision_regions(X_combined, y_combined,
                      classifier=bdt, test_idx=range(105,150))

plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
```



3.2.3 梯度提升决策树 GBDT（Gradient Boosting Decison Tree）

AdaBoost 算法通过给已有模型预测错误的样本更高的权重，使得先前的学习器做错的训练样本在后续受到更多的关注的方式来弥补已有模型的不足。与 AdaBoost 算法不同，梯度提升方法在迭代的每一步构建一个能够沿着梯度最陡的方向降低损失（steepest-descent）的学习器来弥补已有模型的不足。



比如上图中对实际岁数 30 岁进行预测：

- 首先在第一个弱分类器（或第一棵树中）随便用一个年龄比如 20 岁来拟合，然后发现误差有 10 岁；-
- 接下来在第二棵树中，用 6 岁去拟合剩下的损失，发现差距还有 4 岁；
- 接着在第三棵树中用 3 岁拟合剩下的差距，发现差距只有 1 岁了；
- 最后在第四课树中用 1 岁拟合剩下的残差，完美。

最终，四棵树的结论加起来，就是真实年龄 30 岁。

GBDT 模型可以表示为决策树的加法模型：

$$f_M(x) = \sum_{m=1}^{M} T(x;\Theta_m)$$

其中 $T(x;\Theta_m)$ 表示 CART 回归树；Θ_m 为决策树的参数，M 为树的个数。

提升树采用前向分布算法，首先确定初始提升树 $f_0(x)$, 第 m 步的模型是

$$f_m(x) = f_{m-1}(x) + T(x;\Theta_m)$$

通过风险极小化确定下一棵树的参数

$$\Theta_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i;\Theta_m))$$

其中 $L(\cdot)$ 是损失函数。

当采用平方误差损失函数时

$$L(y, f(x)) = (y - f(x))^2$$

损失变为

$$L(y, f_{m-1}(x) + T(x; \Theta_m)) = [y - f_{m-1}(x) - T(x; \Theta_m)]^2 = [r - T(x; \Theta_m)]^2$$

其中 $r = y - f_{m-1}(x)$ 是当前模型拟合数据的残差。所以每一步只需拟合当前模型的残差。

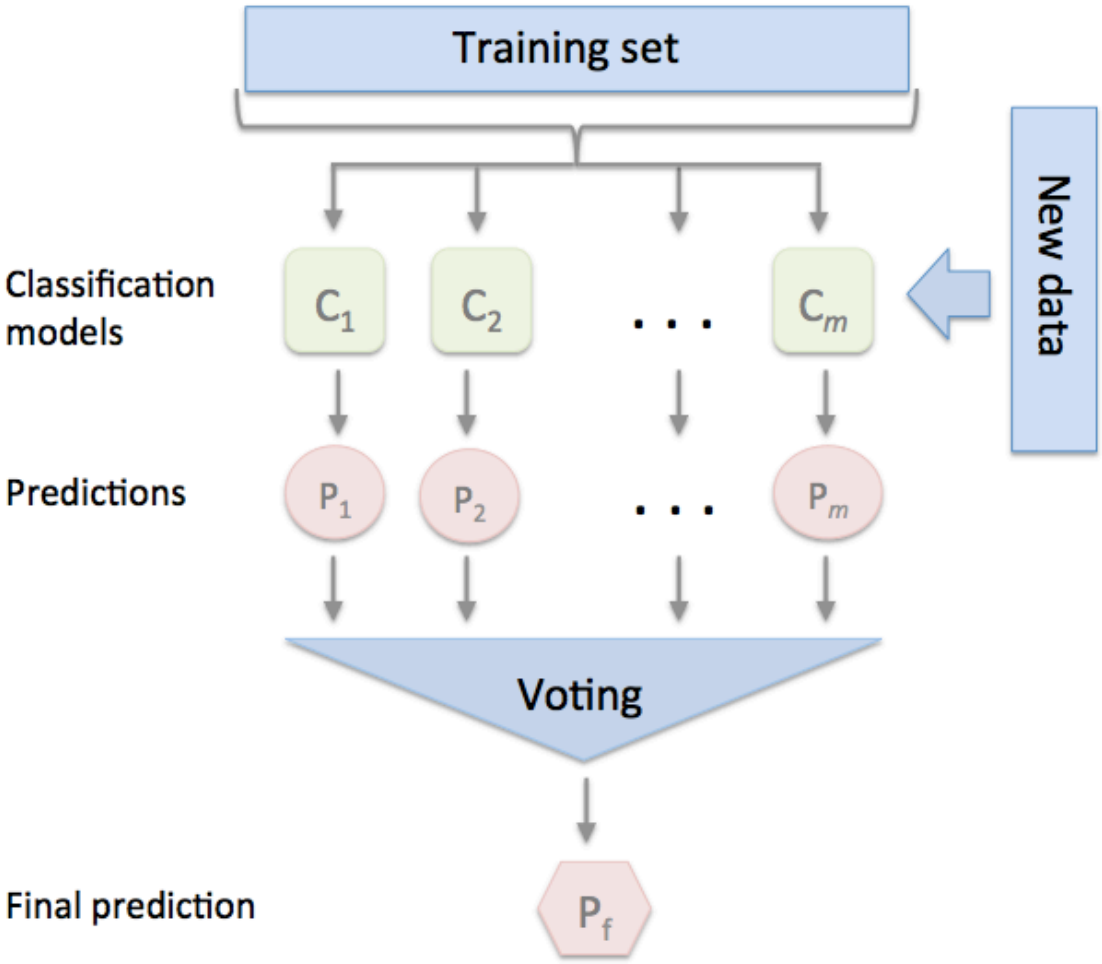
[17]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split

diabetes = datasets.load_diabetes()
# 拆分成训练集和测试集，测试集大小为原始数据集大小的 1/4
X_train,X_test,y_train,y_test = train_test_split(diabetes.data,diabetes.
 ↪target,test_size=0.25,random_state=0)


# 梯度提升决策树 GradientBoostingRegressor 回归模型
regr = GradientBoostingRegressor(n_estimators=100)
regr.fit(X_train,y_train)

# Return the coefficient of determination R^2
print("Training score:%f"%regr.score(X_train,y_train))
```

```
Training score:0.878471
```

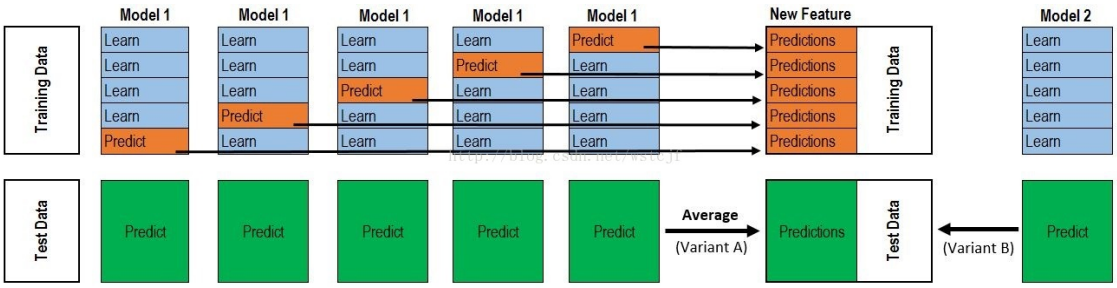
3.3 集成学习之 stacking

3.3.1 原理

Stacking 的主要思想是将训练集弱学习器的学习结果作为输入，将训练集的输出作为输入，重新训练一个学习器来得到最终结果。



下图是一个 5 折 stacking 中基模型在所有数据集上生成预测结果的过程，次学习器会基于模型的预测结果进行再训练，单个基模型生成预测结果的过程是： - 首先将所有数据集生成测试集和训练集（假如训练集为 10000, 测试集为 2500 行），那么上层会进行 5 折交叉检验，使用训练集中的 8000 条作为训练集，剩余 2000 行作为验证集（橙色）- 每次验证相当于使用了蓝色的 8000 条数据训练出一个模型，使用模型对验证集进行验证得到 2000 条数据，并对测试集进行预测，得到 2500 条数据，这样经过 5 次交叉检验，可以得到中间的橙色的 5*2000 条验证集的结果 (相当于每条数据的预测结果)，52500 条测试集的预测结果。 - 接下来会将验证集的 5*2000 条预测结果拼接成 10000 行长的矩阵，标记为 A1，而对于 52500 行的测试集的预测结果进行加权平均，得到一个 2500 一列的矩阵，标记为 B1。 - 上面得到一个基模型在数据集上的预测结果 A1、B1, 这样当我们对 3 个基模型进行集成的话，相于得到了 A1、A2、A3、B1、B2、B3 六个矩阵。



- 之后将 A1、A2、A3 并列在一起成 10000 行 3 列的矩阵作为 training data,B1、B2、B3 合并在一起成 2500 行 3 列的矩阵作为 testing data，让下层学习器基于这样的数据进行再训练。
- 再训练是基于每个基础模型的预测结果作为特征（三个特征），次学习器会学习训练如果往这样的基学习的预测结果上赋予权重 w，来使得最后的预测最为准确。

安装 mlxtend 库，使用它可以方便地对 sklearn 模型进行 stacking。

conda install mlxtend –channel conda-forge

StackingClassifier(classifiers, meta_classifier, use_probas=False, average_probas=False, verbose=0, use_features_in_secondary=False)

参数：

- classifiers： 基分类器，数组形式，[cl1, cl2, cl3]. 每个基分类器的属性被存储在类属性 self.clfs_.
- meta_classifier：目标分类器，即将前面分类器合起来的分类器
- use_probas：bool (default: False)，如果设置为 True，那么目标分类器的输入就是前面分类输出的类别概率值而不是类别标签
- average_probas：bool (default: False)，用来设置上一个参数当使用概率值输出的时候是否使用平均值。
- use_features_in_secondary：bool (default: False). 如果设置为 True，那么最终的目标分类器就被基分类器产生的数据和最初的数据集同时训练。如果设置为 False，最终的分类器只会使用基分类器产生的数据训练。

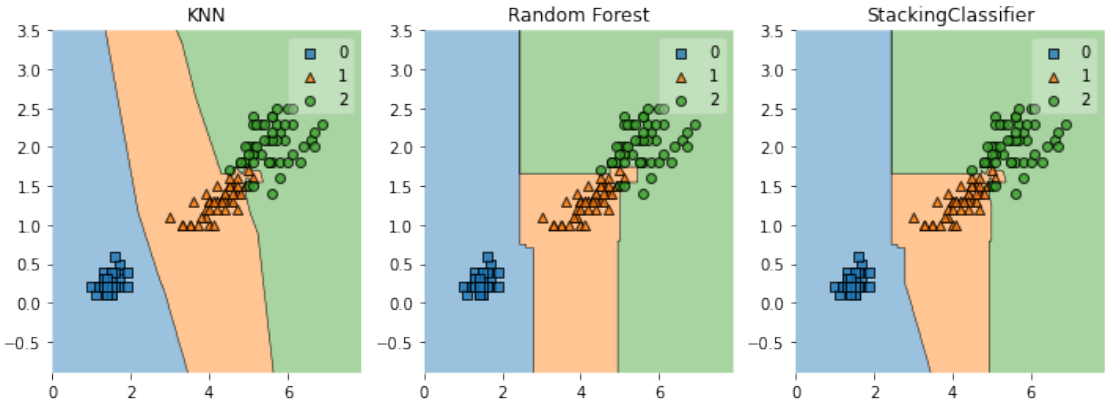
[18]:
```
from sklearn import datasets


iris = datasets.load_iris()
X, y = iris.data[:, [2,3]], iris.target
```

[19]:
```
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier
import numpy as np
import warnings

warnings.simplefilter('ignore')

clf1 = KNeighborsClassifier(n_neighbors=1)
clf2 = RandomForestClassifier(random_state=1)
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2],
                          meta_classifier=lr)

print('3-fold cross validation:\n')

for clf, label in zip([clf1, clf2, sclf],
                      ['KNN',
                       'Random Forest',
                       'StackingClassifier']):

    scores = model_selection.cross_val_score(clf, X, y, cv=3,␣
 ↪scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

```
3-fold cross validation:

Accuracy: 0.95 (+/- 0.04) [KNN]
Accuracy: 0.95 (+/- 0.04) [Random Forest]
Accuracy: 0.95 (+/- 0.04) [StackingClassifier]
```

[20]:
```
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
```

```
import itertools

gs = gridspec.GridSpec(1, 3)

fig = plt.figure(figsize=(12,4))

for clf, lab, grd in zip([clf1, clf2, sclf],
                         ['KNN',
                          'Random Forest',
                          'StackingClassifier'],
                          [(0,0),(0,1),(0,2)]):

    clf.fit(X, y)
    ax = plt.subplot(gs[grd[0], grd[1]])
    fig = plot_decision_regions(X=X, y=y, clf=clf)
    plt.title(lab)
```



3.3.2 GBDT+LR

GBDT+LR 算法是 Facebook 提出的，使用最广泛的场景是 CTR 点击率预估，即预测给用户推送的广告会不会被用户点击。正如它的名字一样，GBDT+LR 由两部分组成，其中 GBDT 用来对训练集提取特征作为新的训练输入数据，Logistic Regression 作为新训练输入数据的分类器。

具体来讲，有以下几个步骤：

- GBDT 首先对原始训练数据做训练，得到一个二分类器，当然这里也需要利用网格搜索寻找最佳参数组合。
- 与通常做法不同的是，当 GBDT 训练好做预测的时候，输出的并不是最终的二分类概率值，而是要把模型中的每棵树计算得到的预测概率值所属的叶子结点位置记为 1，这样，就构造出了新的训练数据。

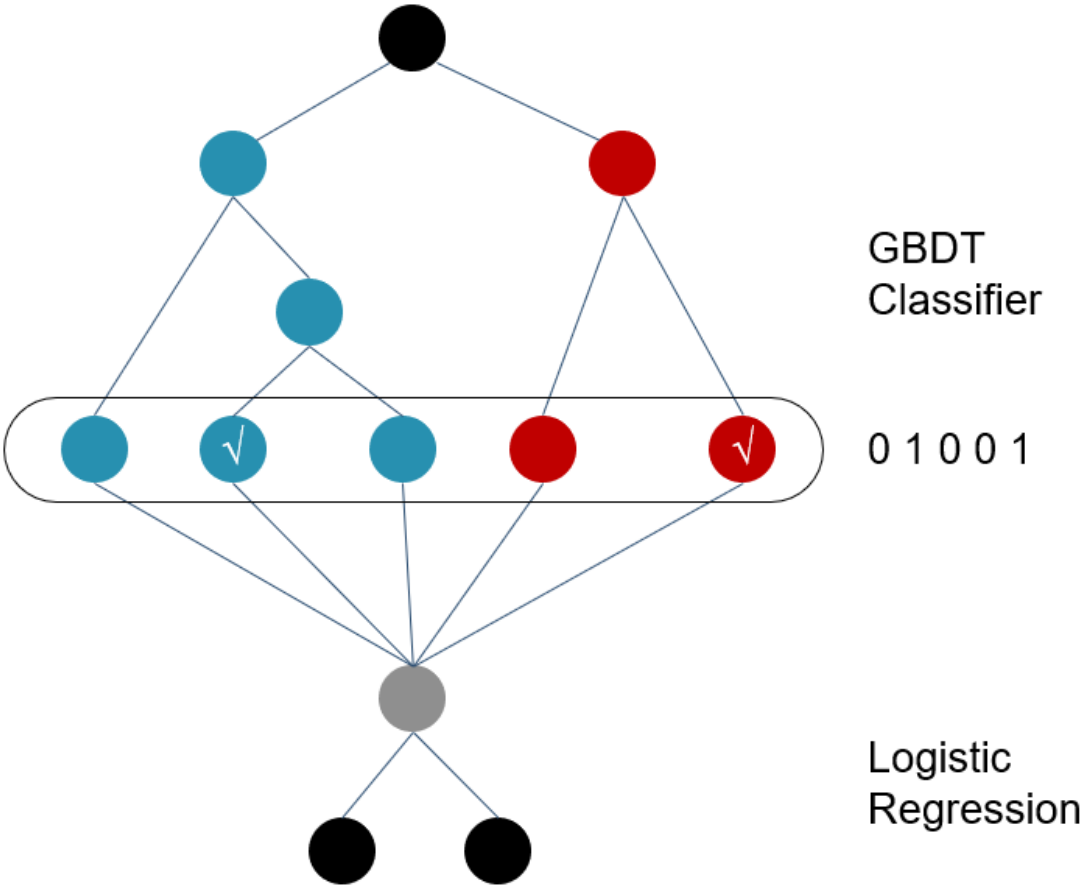
举个例子，下图是一个 GBDT+LR 模型结构，设 GBDT 有两个弱分类器，分别以蓝色和红色部分表示，其中蓝色弱分类器的叶子结点个数为 3，红色弱分类器的叶子结点个数为 2，并且蓝色弱分类器中对 0-1 的预测结果落到了第二个叶子结点上，红色弱分类器中对 0-1 的预测结果也落到了第二个叶子结点上。那么我们就记蓝色弱分类器的预测结果为 [0 1 0]，红色弱分类器的预测结果为 [0 1]，综合起来看，GBDT 的输出为这些弱分类器的组合 [0 1 0 0 1] ，或者一个稀疏向量（数组）。

这里的思想与 One-hot 独热编码类似，事实上，在用 GBDT 构造新的训练数据时，采用的也正是 One-hot 方法。并且由于每一弱分类器有且只有一个叶子节点输出预测结果，所以在一个具有 n 个弱分类器、共计 m 个叶子结点的 GBDT 中，每一条训练数据都会被转换为 1*m 维稀疏向量，且有 n 个元素为 1，其余 m-n 个元素全为 0。

- 新的训练数据构造完成后，下一步就要与原始的训练数据中的 label(输出) 数据一并输入到 Logistic Regression 分类器中进行最终分类器的训练。思考一下，在对原始数据进行 GBDT 提取为新的数据这一操作之后，数据不仅变得稀疏，而且由于弱分类器个数，叶子结点个数的影响，可能会导致新的训练数据特征维度过大的问题，因此，在 Logistic Regression 这一层中，可使用正则化来减少过拟合的风险，在 Facebook 的论文中采用的是 L1 正则化。

建树算法为什么采用 GBDT 而不是 RF?

对于 GBDT 而言，前面的树，特征分裂主要体现在对多数样本的具有区分度的特征；后面的树，主要体现的是经过前面 n 棵树，残差依然比较大的少数样本。优先选用在整体上具有区分度的特征，再选用针对少数样本有区分度的特征, 思路更加合理。

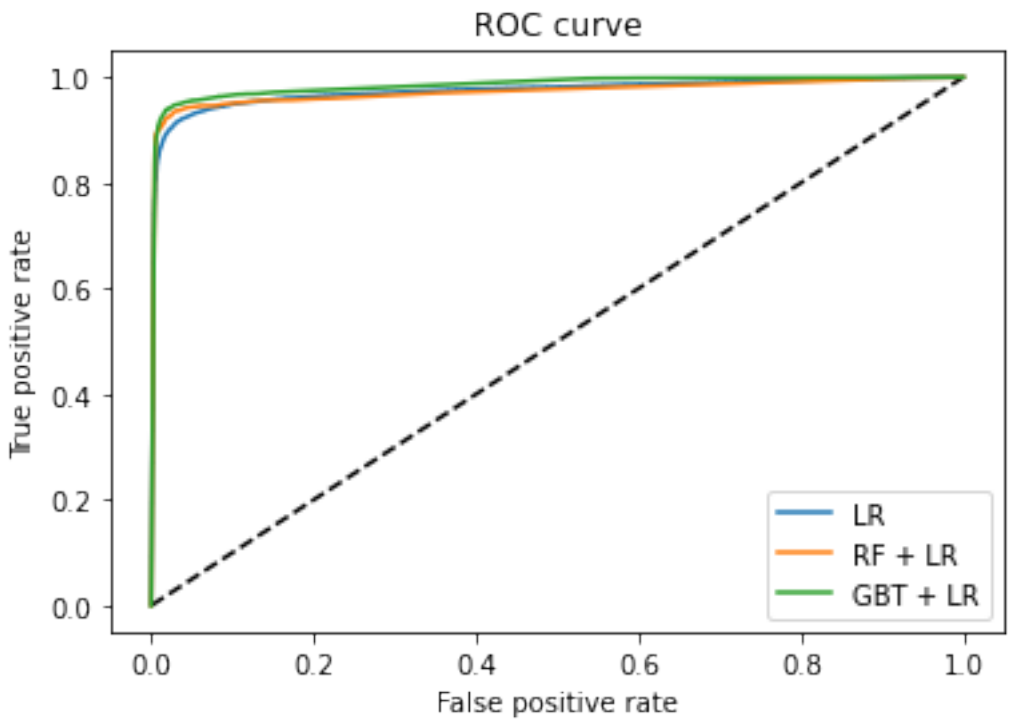


[21]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier,RandomForestClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, roc_auc_score

np.random.seed(10)
n_estimator = 10

X, y = make_classification(n_samples=80000)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
#To avoid overfitting
```

```
X_train, X_train_lr, y_train, y_train_lr = train_test_split(X_train, y_train,␣
↪test_size=0.5)

def Lr():
    lm = LogisticRegression(n_jobs=4, C=0.1)
    lm.fit(X_train, y_train)
    y_lr_test = lm.predict_proba(X_test)[:, 1]
    fpr_lr, tpr_lr, _ = roc_curve(y_test, y_lr_test)
    auc = roc_auc_score(y_test, y_lr_test)
    print("LR:", auc)
    return fpr_lr, tpr_lr

def RandomForestLR():
    rf = RandomForestClassifier(max_depth=3, n_estimators=n_estimator)
    rf_enc = OneHotEncoder()
    # 用默认的 solver 'lbfgs' 不收敛
    rf_lr = LogisticRegression(solver='newton-cg')
    rf.fit(X_train, y_train)
    # model.apply(X_train) 返回训练数据 X_train 在训练好的模型里每棵树中所处的叶子
节点的位置（索引）
    # OneHotEncoder() ,fit() 待转换的数据后，再 transform() 待转换的数据，就可实现
对这些数据的所有特征进行 One-hot 操作。
    rf_enc.fit(rf.apply(X_train))
    rf_lr.fit(rf_enc.transform(rf.apply(X_train_lr)), y_train_lr)
    y_pred_rf_lr = rf_lr.predict_proba(rf_enc.transform(rf.apply(X_test)))[:, 1]
    fpr_rf_lr, tpr_rf_lr, _ = roc_curve(y_test, y_pred_rf_lr)
    auc = roc_auc_score(y_test, y_pred_rf_lr)
    print("RF+LR:", auc)
    return fpr_rf_lr, tpr_rf_lr

def GdbtLR():
    grd = GradientBoostingClassifier(n_estimators=n_estimator)
    grd_enc = OneHotEncoder()
    # 用默认的 solver 'lbfgs' 不收敛
    grd_lr = LogisticRegression(solver='newton-cg')
    grd.fit(X_train, y_train)
```

```
    grd_enc.fit(grd.apply(X_train)[:, :, 0])
    grd_lr.fit(grd_enc.transform(grd.apply(X_train_lr)[:, :, 0]), y_train_lr)
    y_pred_grd_lr = grd_lr.predict_proba(grd_enc.transform(grd.apply(X_test)[:,␣
↪:, 0]))[:, 1]
    fpr_grd_lr, tpr_grd_lr, _ = roc_curve(y_test, y_pred_grd_lr)
    auc = roc_auc_score(y_test, y_pred_grd_lr)
    print("GDBT+LR:", auc)
    return fpr_grd_lr, tpr_grd_lr

if __name__ == '__main__':
    fpr_lr, tpr_lr = Lr()
    fpr_rf_lr, tpr_rf_lr = RandomForestLR()
    fpr_grd_lr, tpr_grd_lr = GdbtLR()



    plt.figure(1)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.plot(fpr_lr, tpr_lr, label='LR')
    plt.plot(fpr_rf_lr, tpr_rf_lr, label='RF + LR')
    plt.plot(fpr_grd_lr, tpr_grd_lr, label='GBT + LR')
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC curve')
    plt.legend(loc='best')
    plt.show()
```

```
LR: 0.9739022156803332
RF+LR: 0.9723935230168325
GDBT+LR: 0.9847114876750664
```



4 集成学习之结合策略

4.1 投票分类

多个弱学习器的对样本 x 的预测结果中，数量最多的类别 c_i 为最终的分类类别。如果不止一个类别获得最高票，则随机选择一个做最终类别。

若每个分类器对应不同的权重 w_j，则每个弱学习器的分类票数要乘以一个权重，最终将各个类别的加权票数求和，最大的值对应的类别为最终类别。

[22]:
```
import numpy as np
np.argmax(np.bincount([0, 0, 1], weights=[0.2, 0.2, 0.6]))


# np.argmax: returns the indices of the maximum values along an axis.
# np.bincount: Count number of occurrences of each value in array of
# non-negative ints
```

[22]: 1

[23]:
```
np.bincount([0, 0, 1], weights=[0.2, 0.2, 0.6])
```

[23]:
```
array([0.4, 0.6])
```

[24]:
```
ex = np.array([[0.9, 0.1],  # C1 的预测结果
               [0.8, 0.2],  # C2 的预测结果
               [0.4, 0.6]]) # C3 的预测结果

# C1, C2, C3 的权重
p = np.average(ex,axis=0,weights=[0.2, 0.2, 0.6])
p
```

[24]:
```
array([0.58, 0.42])
```

$$p(i_0|x) = 0.58$$

$$p(i_1|x) = 0.42$$

$$\hat{y} = \arg\max_i [p(i_0|x), p(i_1|x)] = 0$$

[25]:
```
np.argmax(p) # return index
```

[25]:
```
0
```

4.2 平均法

对于数值类的回归预测问题，通常使用的结合策略是平均法，也就是说，对于若干个弱学习器的输出进行平均得到最终的预测输出。

5 Sklearn 中的投票分类器

[26]:
```
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier

# 使用 3 个分类器
```

```
clf1 = LogisticRegression(random_state=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()

# 生成数据
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])

# voting='hard', 使用预测的 class 作为直接投票
eclf1 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb',␣
 ↪clf3)], voting='hard')
eclf1 = eclf1.fit(X, y)
print(eclf1.predict(X))

# voting='soft', predicts the class label based on the argmax
# of the sums of the predicted probalities
# voting='soft', 基于 argmax 的预测概率的 sum 作为投票
eclf2 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb',␣
 ↪clf3)],voting='soft')
eclf2 = eclf2.fit(X, y)
print(eclf2.predict(X))

# add weight
eclf3 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2), ('gnb',␣
 ↪clf3)],voting='soft', weights=[2,1,1])
eclf3 = eclf3.fit(X, y)
print(eclf3.predict(X))
```

```
[1 1 1 2 2 2]
[1 1 1 2 2 2]
[1 1 1 2 2 2]
```

- hard，对应的就是少数服从多数的投票方式
- soft, 不仅要看投给 A 多少票，B 多少票，还要看到底有多大概率是把样本分成 A/B 类，在使用时，就要求集合的每一个模型都能估计概率。

5.1 对不同分类算法进行投票

[27]:
```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder

# load iris data
iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]

st = StandardScaler()
X = st.fit_transform(X)

le = LabelEncoder()
y = le.fit_transform(y)

# 50% train, 50% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,␣
↪random_state=1)
```

[28]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score

clf1 = LogisticRegression(C=0.01, random_state=42)
clf2 = KNeighborsClassifier(n_neighbors=1)
clf3 = DecisionTreeClassifier(max_depth=1, random_state=42)

clf_labels = ['Logistic Regression', 'KNN', 'Decision Tree']
all_clf = [clf1, clf2, clf3]

print('10-fold cross validation:\n')
# 采样了交叉验证
for clf, label in zip(all_clf, clf_labels):
```

```
    scores = cross_val_score(estimator=clf, X=X_train, y=y_train,cv=10,␣
 ↪scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"% (scores.mean(), scores.std(),␣
 ↪label))
```

```
10-fold cross validation:

ROC AUC: 0.93 (+/- 0.15) [Logistic Regression]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
```

[33]:
```
from sklearn.ensemble import VotingClassifier

mv_clf = VotingClassifier(estimators=[('c1', clf1), ('c2', clf2), ('c3',␣
 ↪clf3)], voting='soft')
clf_labels += ['Majority Voting']
all_clf += [mv_clf]

print('10-fold cross validation:\n')
for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf, X=X_train, y=y_train,cv=10,␣
 ↪scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"% (scores.mean(), scores.std(),␣
 ↪label))
```

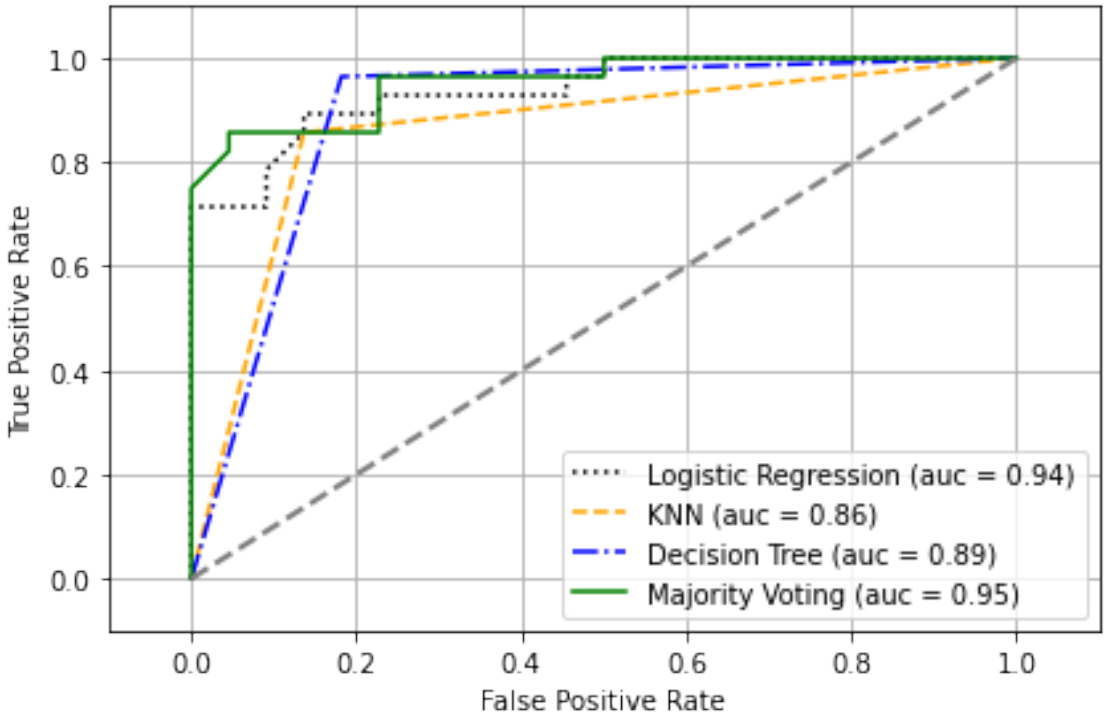
```
10-fold cross validation:

ROC AUC: 0.93 (+/- 0.15) [Logistic Regression]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```

最后一个是 majority voting, 明显比单独的分类器结果好

6 评价集成学习器

6.1 ROC 和 AUC

[30]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
# 模型在 all_clf 里
for clf, label, clr, ls in zip(all_clf, clf_labels, colors, linestyles):

    # assuming the label of the positive class is 1
    # 假设正例的标签是 1
    y_pred = clf.fit(X_train, y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test,y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr,
             color=clr,
             linestyle=ls,
             label='%s (auc = %0.2f)' % (label, roc_auc))

plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1],
         linestyle='--',
         color='gray',
         linewidth=2)

plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

plt.tight_layout()
# plt.savefig('./figures/roc.png', dpi=300)
```

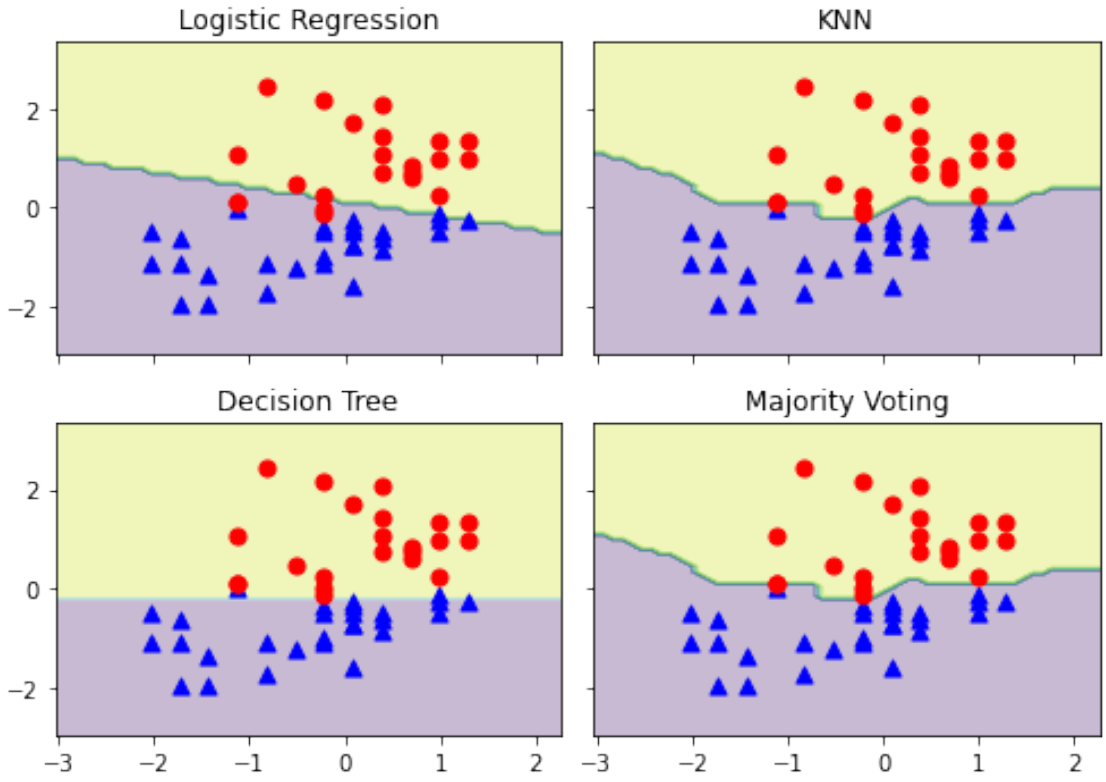


ROC 可看出, ensemble classifier 在 test set 上表现不错

6.2 决策边界

[31]:
```
from itertools import product
import numpy as np

x_min = X_train[:, 0].min() - 1
x_max = X_train[:, 0].max() + 1
y_min = X_train[:, 1].min() - 1
y_max = X_train[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

# 添加 subplots
f, axarr = plt.subplots(nrows=2, ncols=2,
                        sharex='col', sharey='row',
```

```
                        figsize=(7, 5))

for idx, clf, tt in zip(product([0, 1], [0, 1]),all_clf, clf_labels):
    clf.fit(X_train, y_train)
    Z = clf.predict(np.vstack([xx.ravel(), yy.ravel()]).T)
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx[0], idx[1]].scatter(X_train[y_train==0, 0],
                                  X_train[y_train==0, 1],
                                  c='blue', marker='^', s=50)
    axarr[idx[0], idx[1]].scatter(X_train[y_train==1, 0],
                                  X_train[y_train==1, 1],
                                  c='red', marker='o', s=50)
    axarr[idx[0], idx[1]].set_title(tt)

# plt.text(-3.5, -4.5,
#          s='Sepal width [standardized]',
#          ha='center', va='center', fontsize=12)
# plt.text(-10.5, 4.5,
#          s='Petal length [standardized]',
#          ha='center', va='center',
#          fontsize=12, rotation=90)

plt.tight_layout()
# plt.savefig('./figures/voting_panel', bbox_inches='tight', dpi=300)
```



7 常用数据集下载地址

http://archive.ics.uci.edu/ml/index.php