Lesson 4.1 CART分类树的建模流程与sklearn评估器参数详解

CART树是目前机器学习领域内最通用的决策树，并且CART树能够同时解决分类问题和回归问题。本节我们将首先介绍关于CART树在进行分类问题建模时的基本流程，同时详细讲解sklearn中相关评估器的参数情况与使用方法。

In [4]:
```
import numpy as np
from matplotlib import pyplot as plt
```

一、CART决策树的分类流程

从基本原理来说，决策是一种非常通用的模型，在模型训练时可以带入分类变量、也可以带入连续变量，同时在进行预测时既能解决分类问题也能解决回归问题，在介绍CART树的原理时，我们先从简单入手，先考虑自变量都是离散变量的分类预测问题，再逐步拓展连续变量的处理方法和回归类问题的预测方法。

1.CART树的基本生长过程

首先我们来看在特征都为分类变量时、围绕分类问题构建CART的基本过程。

1.1 规则评估指标选取与设置

一般来说树模型挑选分类规则是父节点划分子节点后子节点数据集标签的纯度。

如果一个数据集标签都比较倾向于取得同一个值（不管是都取0还是都取1），则我们就说这个数据集的标签纯度高，反之则说这个这个数据集标签纯度不高。而决策树的划分规则的评估标准，其实都是根据纯度来进行构建的。

> 决策树生长的方向也就是令每个划分出来的子集纯度越来越高的方向。

- 单独一个数据集的标签纯度衡量指标

首先我们先来讨论对于一个单独的数据集来说，可以通过哪些指标来衡量该数据集标签的纯度。一般来说，用于衡量数据集标签纯度的数值指标一般有二种，分别是信息熵和基尼系数，对于每个单独数据集来说，其基本计算公式如下：

(1)信息熵（Entropy）：

$$Entropy(t) = -\sum_{i=1}^{c} p(i|t)log_2 p(i|t)$$

信息熵用于衡量数据混乱程度，也可以用于衡量数据集标签纯度，其计算过程：每个类别的数据占比乘以以2为底的对数处理结果，然后再进行不同类别的累加，最后进行负值处理。例

如针对一个包含6个0类、4个1类的数据集，信息熵计算结果为：

In [87]:
```
- 4/10 * np.log2(4/10) - 6/10 * np.log2(6/10)
```

Out[87]:
```
0.9709505944546686
```

In [88]:
```
def entropy(p):
    """
    信息熵计算函数
    """
    if p == 0 or p == 1:
        ent = 0
    else:
        ent = -p * np.log2(p) - (1-p) * np.log2(1-p)
    return ent
```

信息熵取值在[0,1]之间，并且信息熵越小则说明数据集纯度越高。

> ID3、C4.5、C5.0算法主要使用信息熵进行划分规则的挑选

(2)基尼系数（Gini）：

$$Gini(t) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

基尼系数通过计算1减去每个类别占比的平方和来作为纯度衡量指标，例如针对一个包含了6个0类、4个1类的数据集，基尼系数的计算结果为：

In [89]:
```
1 - np.power([4/10, 6/10], 2).sum()
```

Out[89]:
```
0.48
```

而和信息熵不同的是，基尼系数在[0, 0.5]范围内取值，并且基尼系数越小表示数据集标签纯度越高

> 在默认情况下，CART树默认选择Gini系数作为评估指标。

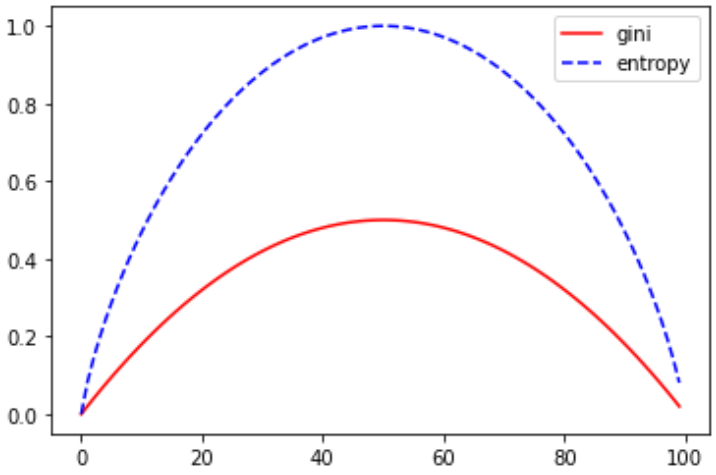
我们可以简单对比在二分类情况下二个评估指标伴随某类样本所占比例p值变化而变化的情况：

In [90]:
```
p_l = np.arange(0, 1, 0.01)
gini_l = []
en_l = []

for p in p_l:
    gini_l.append(1 - np.power([p, 1-p], 2).sum())
    en_l.append(entropy(p))
```

In [91]:
```
plt.plot(gini_l, 'r-', label='gini')
plt.plot(en_l, 'b--', label='entropy')
plt.legend()
```

Out[91]:
```
<matplotlib.legend.Legend at 0xd3ca9c8>
```



1.2 决策树备选规则创建方法

对于很多树模型，特征是离散型变量还是连续性变量会直接影响备选规则的创建，但对于CART树以及slearn中集成的方法来说，是将两种情况合二为一来进行看待，也就是根据如下方式来进行节点划分阈值的创建：

举例：简化后客户信用数据集A共有两个特征、一个标签，并且标签只有0-1两个类别，数据集特征分别是收入（income）和信用评级（credit_rating），同样也都用有两个分类水平的离散变量表示。

数据集A			
ID	**income**	**credit_rating**	**class**
1	1	1	0
2	2	2	0
3	2	1	0
4	1	2	1
5	1	1	0
6	1	2	1
7	1	2	1
8	2	1	0

对于任何特征矩阵，首先需要逐列对其进行数值排序，例如上述数据集A，我们可以单独提取income和credit_rating两列来进行降序排序，排序结果如下：

ID	**2**	**3**	**8**	**1**	**4**	**5**	**6**	**7**
income	2	2	2	1	1	1	1	1

ID	**2**	**4**	**6**	**7**	**1**	**3**	**5**	**8**
credit_rating	2	2	2	2	1	1	1	1

据此，我们通过寻找这些特征不同取值之间的中间点作为切点，来构造分支条件。例如income有两个取值1、2，因此只有一个切点就是1.5，那么我们就能创造一个income <= 1.5的规则来对数据集进行划分，如此我们就能把income取值为1的数据划归一个子集、income取值为2的数据集划归另一个子集，实际上上面在介绍多数据集基尼系数计算过程时就是采用该规则。需要知道的是，在所构造的条件中不等号的方向实际上没有任何影响。当然，income只有两个取值只能找到一个切点只能构造一个规则，而credit_rating特征也有两个取值，因此也能找到一个切点构造一个备选规则，即我们其实也可以根据credit_rating <= 1.5来切分出两个子集。

而其实如果特征是三个取值的特征，则可以找到两个切点、找到两种划分数据集的方式。更进一步的，如果该数据中某特征是连续变量，每条不同的数据取值都不同，例如：

ID	1	3	7	2	4	6	5	8
age	37	35	32	26	24	19	17	15

则此时可以将其看成是拥有8个分类水平的分类变量，仍然还是寻找相邻取值水平的中间值作为切点来构造划分规则，此时由于age特征有8个不同的取值，因此可以构造7个备选的划分数据集的方法，例如age <= 36、age <= 34.5等等。也就是说，对于任何一个特征无论是连续型变量还是分类变量，只要有N个取值，就可以创造N-1个划分条件将原数据集划分成两份。

> 此外，需要注意的是，CART树用这种方法同时处理离散变量和连续变量，而C4.5只用这种方式处理连续变量（离散变量采用另一种方法），因此这里我们可以理解成是CART树将离散变量“连续化”，也就是将离散变量看成是连续变量，这也就是为何sklearn在说明文档中强调，sklearn的树模型无法处理分类变量的原因（原文：scikit-learn implementation does not support categorical variables for now.）。此处所谓的无法处理分类变量并不是不能带入分类变量进行建模，而是不支持类似C4.5的方法从离散特征中提取备选划分规则，而是会将离散变量也看成是连续变量。

1.3 挑选最佳分类规则划分数据集

对于所有的划分规则，采用上述方法进行数据集不纯度的衡量。然后选择不纯度下降（信息增益和gini指数）最多的分支特征和阈值，作为当前最优的划分规则。

此时我们首先可以计算该数据集整体的基尼系数：

In [92]:
```
y = np.array([0,0,0,1,0,1,1,0])
A = np.array([[1,1],[2,2],[2,1],[1,2],[1,1],[1,2],[1,2],[2,1]])
p = np.mean(y)  # p = 3/8
gini_A = 1 - np.power([p, 1-p], 2).sum()
entroy_A = entropy(p)
p, gini_A , entroy_A
```

Out[92]:
```
(0.375, 0.46875, 0.954434002924965)
```

然后我们随意设置一个分类条件。例如我们设置分类条件为income <= 1.5，则可以将上述数据集进一步划分成两个子数据集B1、B2:

数据集A			
ID	income	credit_rating	class
1	1	1	0
2	2	2	0
3	2	1	0
4	1	2	1
5	1	1	0
6	1	2	1
7	1	2	1
8	2	1	0

TRUE　　划分条件：income <= 1.5　　FALSE

数据集B1			
ID	income	credit_rating	class
1	1	1	0
4	1	2	1
5	1	1	0
6	1	2	1
7	1	2	1

数据集B2			
ID	income	credit_rating	class
2	2	2	0
3	2	1	0
8	2	1	0

In [93]:
```
index = A[:,0]<1.5
index
```

Out[93]:
```
array([ True, False, False,  True,  True,  True,  True, False])
```

In [94]:
```
B1 = A[A[:,0]<= 1.5]
B1
```

Out[94]:
```
array([[1, 1],
       [1, 2],
       [1, 1],
       [1, 2],
       [1, 2]])
```

In [95]:
```
y_B1 = y[A[:,0]<1.5]
```

In [96]:
```
p1 = np.mean(y_B1)  # p = 2/5
gini_B1 = 1 - np.power([p, 1-p], 2).sum()
gini_B1
```

Out[96]:
```
0.46875
```

而B2数据集只包含一个标签，因此B2的基尼系数为0

In [97]:
```
gini_B2 = 0
```

而此时如果要计算B1、B2整体的基尼系数，则需要在gini_B1、gini_B2的基础上进行各自数据集样本数量占整体数据集比例的加权求和，即根据如下方式进行计算：

$$Gini(B) = \frac{|B_1|}{|A|}Gini(B_1) + \frac{|B_2|}{|A|}Gini(B_2)$$

其中$\frac{|B_i|}{|A|}$为子数据集B_i数据个数占父类数据集A中数据个数的比例。

因此上述B1、B2整体基尼系数为：

In [98]:
```
len(y_B1)/len(y)
```

Out[98]: 0.625

In [99]:
```
# gini_B = gini_B1 * 5/8 + gini_B2 * 3/8
gini_B = gini_B1 * (len(y_B1)/len(y)) + gini_B2 * (1-len(y_B1)/len(y))
gini_B
```

Out[99]: 0.29296875

In [100…
```
# 计算该分支的gini指数下降量
gini_A - gini_B
```

Out[100]: 0.17578125

而如果我们采用credit_rating <= 1.5来对数据集进行划分，则将出现下述结果：

数据集A

ID	income	credit_rating	class
1	1	1	0
2	2	2	0
3	2	1	0
4	1	2	1
5	1	1	0
6	1	2	1
7	1	2	1
8	2	1	0

TRUE　　FALSE

划分条件：credit_rating <= 1.5

数据集B1

ID	income	credit_rating	class
1	1	1	0
3	2	1	0
5	1	1	0
8	2	1	0

数据集B2

ID	income	credit_rating	class
2	2	2	0
4	1	2	1
6	1	2	1
7	1	2	1

以income <= 1.5为规则划分数据集时，基尼系数下降结果为：

In [101…
```
p = 3/4
gini_B2 = 1 - np.power([p, 1-p], 2).sum()
gini_B2
```

Out[101]: 0.375

In [102…
```
gini_B1 = 0
```

In [103…
```
gini_B = gini_B1 * 1/2 + gini_B2 * 1/2
gini_B
```

Out[103]: 0.1875

In [104…
```
gini_A - gini_B
```

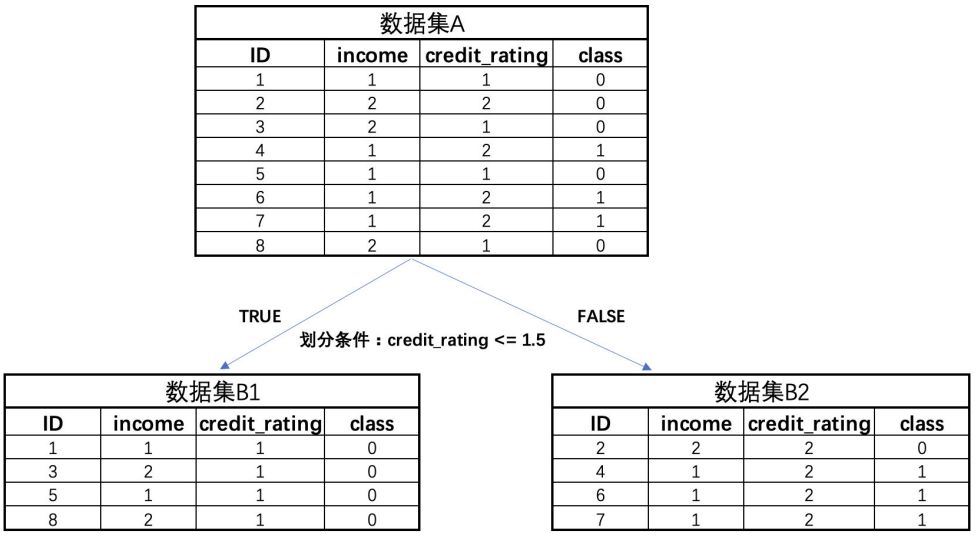
Out[104]: 0.28125

很明显，第二个规则能够让父节点的基尼系数下降更快，因此第二个规则、即credit_rating <= 1.5划分规则是一个更好的规则，在第一次数据集划分时我们应该采用该规则。

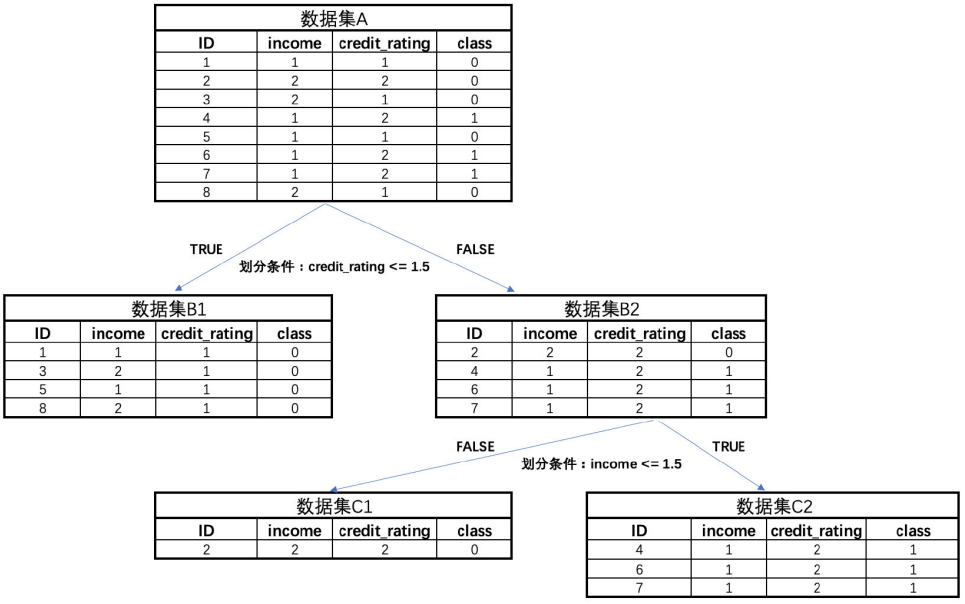
> 注，如果是ID3或者C4.5，此处则是以信息熵计算结果为准。

1.4 决策树的生长过程

当完成一次规则筛选与树生长后，接下来需要考虑的问题是，面对当前划分出的数据集B1、B2，是否还需要进一步寻找分类规则对其进行划分。



首先，对于数据集B1来说，由于其基尼系数已经为0，无需再进行计算；而B2数据集基尼系数为0.375，还可以进一步提取有效分类规则对其进行分类，以降低其基尼系数。此时我们又可以完全重复数据集A的划分过程，首先围绕数据集B2进行备选规则的提取，对于B2来说备选规则只有income <= 1.5一条，因此我们就以该规则划分数据集：



能够看出，最终划分出来的C1和C2基尼系数都是0，因此C的两个数据集整体基尼系数也是0，当然我们也无需进一步划分数据集，到此为止决策树也停止生长。

- 决策树生长与迭代运算

决策树的生长过程本质上也是在进行迭代运算，我们根据上一轮的到的结论（数据集划分情况）作为基础条件，来寻找子数据集的最佳分类规则，然后来进行数据集划分，以此往复。

迭代计算的终止条件。最直接的就是叶子节点的不纯度为0。也可以是基尼系数的减少少于

某个值就暂时不做划分、或者已到达到树的最大设置深度、或叶子节点数量少于某个值就暂时不做划分等。

2.CART树的剪枝

和逻辑回归不同，决策树在不进行特征衍生时就是一个分类效力更强的模型，因此其本身就是一个更容易过拟合的模型。并且决策树生长的层数越多树模型越复杂，此时模型结构风险就越高、模型越容易过拟合。因此，很多时候如果我们不对树的生长进行任何约束，即如果设置的收敛条件较为严格（例如要求最终基尼系数全为0），并且最大迭代次数不进行限制，则很有可能容易过拟合。因此在决策树的建模流程当中，有非常重要的一个环节，就是需要讨论限制决策树模型过拟合倾向的方法——剪枝策略。

当然，不同决策树算法的剪枝策略也各有不同，总的来说树模型的剪枝分为两种，其一在模型生长前就限制模型生长，这种方法也被称为预剪枝或者盆栽法；而另外一种方法则是先让树模型尽可能的生长，然后再进行剪枝，这种方法也被称为后剪枝或者修建法。从算法的原生原理来讲，目前主流的C4.5和CART树都采用的是后剪枝的方法。

不过，无论采用何种方式来进行剪枝，最终的结果都是通过控制树的结构复杂度来抑制过拟合倾向，而树模型的结构复杂度其实完全可以用树的层数、每一层分叉的节点数来表示，即内部节点和叶节点的数量来表示，我们直接将这些决定树模型的复杂度的因素视作超参数，然后通过网格搜索的方式来直接确定泛化能力最强的树模型结构。当然这也是sklearn中进行决策树剪枝的一般策略。

二、CART分类树的Scikit-Learn快速实现方法与评估器参数详解

1.CART分类树的sklearn快速实现

接下来我们尝试在Scikit-Learn中构建分类树模型。在sklearn中，回归树和分类树是两个不同的评估器，都在sklearn.tree模块内，我们可以通过如下方式进行导入：

In [105…
```
from sklearn.tree import DecisionTreeClassifier,DecisionTreeRegressor
```

然后尝试围绕上述简单例子进行快速建模试验：

In [106…
```
# 准备数据集
X = np.array([[1, 1], [2, 2], [2, 1], [1, 2], [1, 1], [1, 2], [1, 2], [2, 1]])
y = np.array([0, 0, 0, 1, 0, 1, 1, 0])
```

In [107…
```
# 调用决策树评估器并进行训练
clf = DecisionTreeClassifier().fit(X, y)
```

In [108…
```
clf.score(X, y)
```

Out[108]:
```
1.0
```

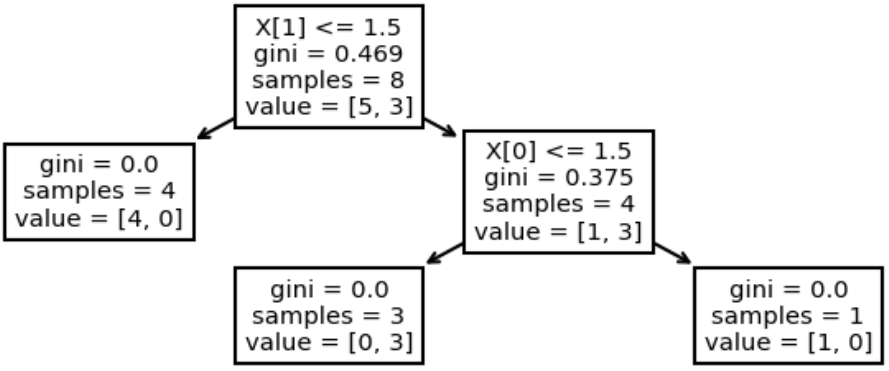
当然，对于树模型来说，我们不仅需要查看模型最终结果的评估指标，很多时候还希望能够观察到树模型分类过程的树状图，可以借助sklearn.tree模块下的plot_tree函数直接输入训练好的评估器即可进行绘制：

- plot_tree绘制树状图

In [109…
```
# 首先导入tree模块
from sklearn import tree
```

In [110…
```
# 然后调用plot_tree函数进行绘制
plt.figure(figsize=(6, 2), dpi=150)
tree.plot_tree(clf)
```

Out[110]:
```
[Text(0.4, 0.8333333333333334, 'X[1] <= 1.5\ngini = 0.469\nsamples = 8\nvalue = [5, 3]'),
 Text(0.2, 0.5, 'gini = 0.0\nsamples = 4\nvalue = [4, 0]'),
 Text(0.6, 0.5, 'X[0] <= 1.5\ngini = 0.375\nsamples = 4\nvalue = [1, 3]'),
 Text(0.4, 0.16666666666666666, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]'),
 Text(0.8, 0.16666666666666666, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]')]
```



由于plot_tree是sklearn中已经集成好的函数，因此调用过程非常简单，我们只需要输入训练好的分类树评估器即可。同时根据输出的结果可知，sklearn中分类树的建模过程和此前我们手动实现的过程是一样的，先根据第一个特征的不同取值进行数据集划分，然后在根据第二个特征的不同取值进行数据集划分，最终形成一个三个叶节点、两层的决策树模型。

2.CART分类树评估器的参数详解

实际上DecisionTreeClassifier评估器参数众多，并且大多和决策树的模型结构相关：

Name	Description
criterion	规则评估指标或损失函数，默认基尼系数，可选信息熵
splitter	树模型生长方式，默认以损失函数取值减少最快方式生长，可选随机根据某条件进行划分
max_depth	树的最大生长深度，类似max_iter，即总共迭代几次
min_samples_split	内部节点再划分所需最小样本数
min_samples_leaf	叶节点包含最少样本数
min_weight_fraction_leaf	叶节点所需最小权重和
max_features	在进行切分时候最多带入多少个特征进行划分规则挑选
random_state	随机数种子
max_leaf_nodes	叶节点最大个数
min_impurity_decrease	数据集再划分至少需要降低的损失值
class_weight	各类样本权重

Name	Description
ccp_alpha	在执行CART树原生原理中的剪枝流程时结构复杂度惩罚因子的系数，默认情况下不使用该方法进行剪枝

接下来围绕一些重点参数进行详细讲解：

- criterion：不纯度衡量指标

首先，我们发现尽管sklearn的树模型在默认情况下是CART树，但同样支持使用信息熵来衡量不纯度。不过需要注意的是，哪怕我们在criterion参数中选择信息熵，实际树模型的建模过程也不是按照ID3或者C4.5的流程执行，此时的树模型只能算是一种混合模型。而关于到底应该选择哪个指标来衡量数据集的不纯度，其实大多数情况下选择哪个指标并不会实质影响树模型的结构，但相比信息熵，基尼系数复杂度更低、计算速度更快，一般情况下推荐使用基尼系数。而如果一定要寻找二者在使用上的不同，一般认为在有些情况下，基尼不纯度更倾向于在数据集中分割出多数类，而信息熵则更倾向于生成出更加平衡的树。

- ccp_alpha：结构风险权重

ccp是复杂度剪枝（Cost-Complexity Pruning）的简称，这是一个在sklearn的0.22版本中才加入的参数，这也是唯一一个为实现CART原生原理中的剪枝过程所设置的参数。此处首先需要知道的是在sklearn中并不一定要通过该方法进行剪枝，因此该参数其实也并不是一个必选参数。其次，带有ccp项的剪枝也被称为最小复杂度剪枝，其原理是在决策树的损失函数上加上一个结构风险项，类似于正则化项在线性方程的损失函数中作用。 我们可以设T为某决策树，R(T)为决策树在训练集上整体不纯度，即代表模型的经验风险，令a|T|表示模型结构风险，其中a为参数，|T|为树的叶节点数量，则我们可以修改模型损失函数如下：

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|$$

a参数是风险结构项的系数，其取值越大、对模型的结构风险惩罚力度就越大、模型结构就越简单、过拟合就能够被更好的抑制，反之亦反。

- 控制树结构的参数簇

接下来就是关于控制树模型结构的相关参数，同时这也是最多的一类参数。这类参数可以进一步细分成两类，其一是限制模型整体结构，主要包括限制树深度的max_depth参数和限制叶节点数量的max_leaf_nodes参数。此外第二类就是限制树生长的参数，包括从节点样本数量限制树生长的参数，包括min_samples_split、min_samples_leaf两个参数，当然也有从损失值降低角度出发限制树生长的参数，包括min_impurity_split和min_impurity_decrease参数。通过这些参数的共同作用，可以从各角度有效限制树的生长。

> 注意，所谓树的最大深度，指的是树的最多生长几层，或者除了根节点外总共有几层，并不是树的总共的层数。

此处需要重点说明的是，对于树模型来说，叶节点太多、单独叶节点所包含的样本数量太少、内部节点再划分降低的基尼系数较少，都是可能是过拟合的表现，在建模时尤其需要注意。

并且需要知道的是，sklearn中在计算父节点和子节点的基尼系数（或信息熵）的差值时，会在计算结果的前面乘以一个父节点占根节点数据量比例的系数作为最终impurity_decrease的结果：

```
"""Compute the improvement in impurity.

This method computes the improvement in impurity when a split occurs.
The weighted impurity improvement equation is the following:

    N_t / N * (impurity - N_t_R / N_t * right_impurity
                        - N_t_L / N_t * left_impurity)

where N is the total number of samples, N_t is the number of samples
at the current node, N_t_L is the number of samples in the left child,
and N_t_R is the number of samples in the right child,
```

这会导致样本比较少的某节点，哪怕再划分时子节点不纯度降低更多，但由于当前节点样本较少，因此impurity_decrease数值较低。这其实也是一种为了防止过拟合而采取的措施。

- 控制迭代随机过程的参数簇

最后，还有一类参数值得注意，那就是关于控制建模过程中随机性的一些参数，主要包含两个，其一是splitter参数，当该参数取值为random时其实是随机挑选分类规则对当前数据集进行划分，其二是max_features，该参数可以任意设置最多带入几个特征进行备选规律挖掘，只要该参数的设置不是带入全部特征进行建模，就相当于是给备选特征随机划个范围，也相当于是给树模型的训练增加了一定的随机性。当然，这两个参数的主要作用有两个方面，其一是可以提升模型训练速度，如果我们只从个别特征中挑选最佳划分规则，或者随机生成一个划分规则、不进行比较就直接使用，其实都能够极大节省计算量，只不过这也是一种用精度换效率的方式，如此操作肯定会带来模型结果精度的下降；不过随机性其实也是一把双刃剑，在集成学习中，为了让各基础分类器“和而不同”，就必须让每个基分类器保证一定的随机性，而决策树就是最常作为基分类器参与到集成学习中的模型，因此树模型中的这些控制其随机性的参数，也会在集成学习中发挥作用。

> 需要注意的是，即使splitter设置为“best”,max_features使用全部特征，决策树仍具有一定的随机性，就是当分支条件得到的不纯度下降相同时，会从可选择的分支条件中随机选择一个作为分支规则，因此，可以通过设置random_state将决策树的随机性确定下来。

In [1]:
```
# Scikit-Learn相关模块
# 评估器类
from sklearn.linear_model import LogisticRegression
from sklearn import tree

# 数据准备
from sklearn.datasets import load_iris
```

In [2]:
```
# 数据准备
# X, y = load_iris(return_X_y=True)
iris = load_iris()
X = iris.data
y = iris.target
```

In [5]:
```
t = np.array(iris.target)
```

```
t[50:]
```

Out[5]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

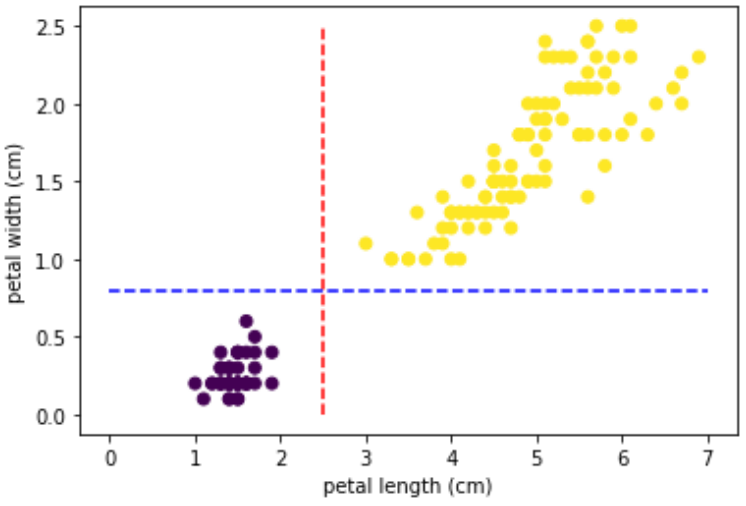
In [6]:
```
# 将2、3类划归为一类
t[50:] = 1
t
```

Out[6]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

In [7]:
```
# 此处提取第3、4个特征放置二维空间进行观察，用第三个特征和其他特征组合也是类似
d = np.array(iris.data[:, 2: 4])
```

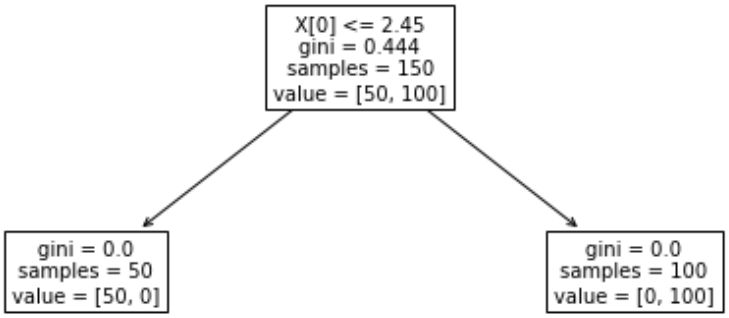
In [8]:
```
plt.scatter(d[:, 0], d[:, 1], c=t)
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[3])
plt.plot(np.array([2.5]*25), np.linspace(0, 2.5, 25), 'r--')
plt.plot(np.linspace(0, 7, 25), np.array([0.8]*25), 'b--')
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0xc525388>]
```



In [14]:
```
clf = tree.DecisionTreeClassifier() #决策树随机选择最优分支
clf = clf.fit(d, t)
plt.figure(figsize=(10,4))
tree.plot_tree(clf,fontsize=10)
clf.feature_importances_
```

Out[14]:
```
array([1., 0.])
```



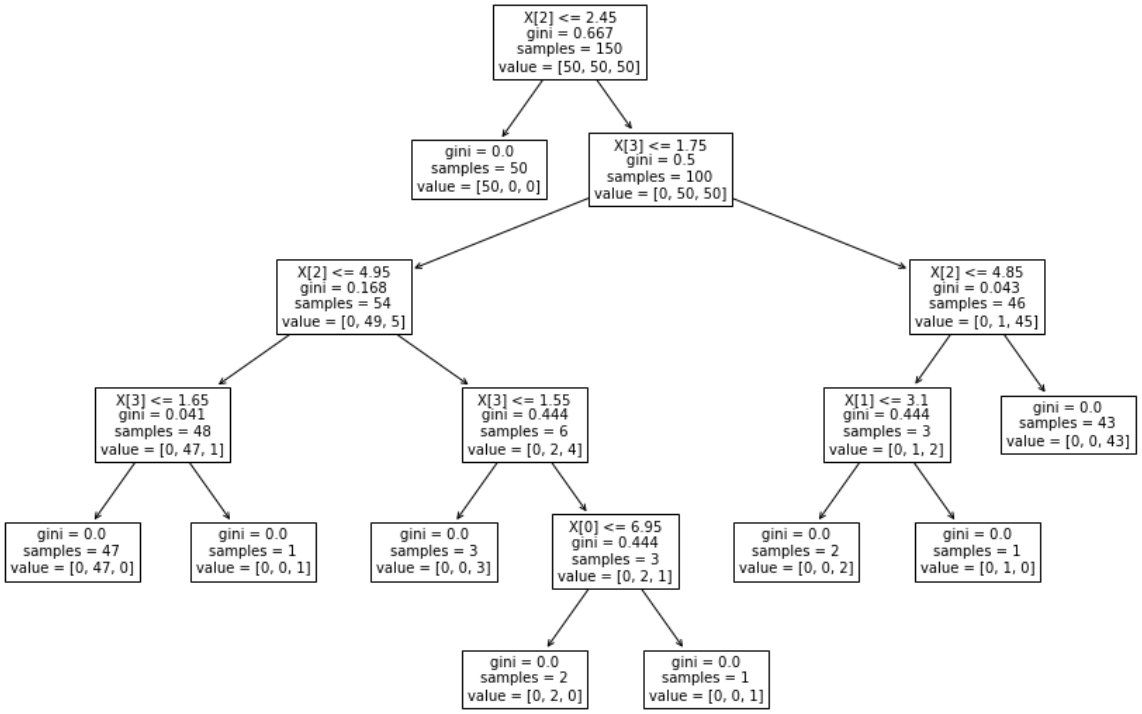
> 当我们选择不同的分支规则之后，决策树的feature_importances_也会随之变化，因为在sklearn当中，feature_importances_是特征对criterion下降量的总贡献量，因此不同的分支规则可能得到不同的特征重要性。

In [15]:
```
list(zip(iris.feature_names,clf.feature_importances_))
```

Out[15]:
```
[('sepal length (cm)', 1.0), ('sepal width (cm)', 0.0)]
```

In [16]:
```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)  #使用全部数据建立树模型
plt.figure(figsize=(15,10))
tree.plot_tree(clf,fontsize=10)
```

Out[16]:
```
[Text(0.5, 0.9166666666666666, 'X[2] <= 2.45\ngini = 0.667\nsamples = 150\nvalue = [50, 50, 50]'),
 Text(0.4230769230769231, 0.75, 'gini = 0.0\nsamples = 50\nvalue = [50, 0, 0]'),
 Text(0.5769230769230769, 0.75, 'X[3] <= 1.75\ngini = 0.5\nsamples = 100\nvalue = [0, 50, 50]'),
 Text(0.3076923076923077, 0.5833333333333334, 'X[2] <= 4.95\ngini = 0.168\nsamples = 54\nvalue = [0, 49, 5]'),
 Text(0.15384615384615385, 0.4166666666666667, 'X[3] <= 1.65\ngini = 0.041\nsamples = 48\nvalue = [0, 47, 1]'),
 Text(0.07692307692307693, 0.25, 'gini = 0.0\nsamples = 47\nvalue = [0, 47, 0]'),
 Text(0.23076923076923078, 0.25, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(0.46153846153846156, 0.4166666666666667, 'X[3] <= 1.55\ngini = 0.444\nsamples = 6\nvalue = [0, 2, 4]'),
 Text(0.38461538461538464, 0.25, 'gini = 0.0\nsamples = 3\nvalue = [0, 0, 3]'),
 Text(0.5384615384615384, 0.25, 'X[0] <= 6.95\ngini = 0.444\nsamples = 3\nvalue = [0, 2, 1]'),
 Text(0.46153846153846156, 0.08333333333333333, 'gini = 0.0\nsamples = 2\nvalue = [0, 2, 0]'),
 Text(0.6153846153846154, 0.08333333333333333, 'gini = 0.0\nsamples = 1\nvalue = [0, 0, 1]'),
 Text(0.8461538461538461, 0.5833333333333334, 'X[2] <= 4.85\ngini = 0.043\nsamples = 46\nvalue = [0, 1, 45]'),
 Text(0.7692307692307693, 0.4166666666666667, 'X[1] <= 3.1\ngini = 0.444\nsamples = 3\nvalue = [0, 1, 2]'),
 Text(0.6923076923076923, 0.25, 'gini = 0.0\nsamples = 2\nvalue = [0, 0, 2]'),
 Text(0.8461538461538461, 0.25, 'gini = 0.0\nsamples = 1\nvalue = [0, 1, 0]'),
 Text(0.9230769230769231, 0.4166666666666667, 'gini = 0.0\nsamples = 43\nvalue = [0, 0, 43]')]
```



In [49]:
```
X.shape
```

Out[49]:
```
(150, 4)
```

In [51]:
```
clf.apply(X)
```

Out[51]:
```
array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  5,
        5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,
        5,  5, 15,  5,  5,  5,  5,  5,  5, 10,  5,  5,  5,  5,  5, 10,  5,
        5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5, 16, 16,
       16, 16, 16, 16,  6, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
        8, 16, 16, 16, 16, 16, 16, 14, 16, 16, 11, 16, 16, 16,  8,  8, 16,
       16, 16, 14, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16],
      dtype=int64)
```

In [37]:
```
clf.classes_
```

Out[37]:
```
array([0, 1, 2])
```

In [20]:
```
clf.tree_.capacity
```

Out[20]:
```
17
```

In [21]:
```
clf.tree_.node_count
```

Out[21]:
```
17
```

In [22]:
```
clf.tree_.max_depth
```

Out[22]:
```
5
```

In [23]:
```
clf.tree_.children_left
```

Out[23]:
```
array([ 1, -1,  3,  4,  5, -1, -1,  8, -1, 10, -1, -1, 13, 14, -1, -1, -1],
      dtype=int64)
```

In [24]:
```
clf.tree_.children_right
```

Out[24]:
```
array([ 2, -1, 12,  7,  6, -1, -1,  9, -1, 11, -1, -1, 16, 15, -1, -1, -1],
      dtype=int64)
```

In [25]:
```
clf.tree_.feature
```

Out[25]:
```
array([ 2, -2,  3,  2,  3, -2, -2,  3, -2,  0, -2, -2,  2,  1, -2, -2, -2],
      dtype=int64)
```

In [26]:
```
clf.tree_.threshold
```

Out[26]:
```
array([ 2.44999999, -2.        ,  1.75      ,  4.95000005,  1.65000004,
       -2.        , -2.        ,  1.55000001, -2.        ,  6.94999981,
       -2.        , -2.        ,  4.85000014,  3.10000002, -2.        ,
       -2.        , -2.        ])
```

In [27]:
```
clf.tree_.impurity
```

Out[27]:
```
array([0.66666667, 0.        , 0.5       , 0.16803841, 0.04079861,
       0.        , 0.        , 0.44444444, 0.        , 0.44444444,
       0.        , 0.        , 0.04253308, 0.44444444, 0.        ,
       0.        , 0.        ])
```

In [28]:
```
clf.tree_.n_node_samples
```

Out[28]:
```
array([150,  50, 100,  54,  48,  47,   1,   6,   3,   3,   2,   1,  46,
         3,   2,   1,  43], dtype=int64)
```

In [29]:
```
clf.tree_.weighted_n_node_samples
```

Out[29]:
```
array([150.,  50., 100.,  54.,  48.,  47.,   1.,   6.,   3.,   3.,   2.,
         1.,  46.,   3.,   2.,   1.,  43.])
```

In [32]:
```
clf.tree_.value
```

Out[32]:
```
array([[[50., 50., 50.]],

       [[50.,  0.,  0.]],

       [[ 0., 50., 50.]],

       [[ 0., 49.,  5.]],

       [[ 0., 47.,  1.]],

       [[ 0., 47.,  0.]],

       [[ 0.,  0.,  1.]],

       [[ 0.,  2.,  4.]],

       [[ 0.,  0.,  3.]],

       [[ 0.,  2.,  1.]],

       [[ 0.,  2.,  0.]],

       [[ 0.,  0.,  1.]],

       [[ 0.,  1., 45.]],

       [[ 0.,  1.,  2.]],

       [[ 0.,  0.,  2.]],

       [[ 0.,  1.,  0.]],

       [[ 0.,  0., 43.]]])
```

In [34]:
```
clf.tree_.n_leaves
```

Out[34]:
```
9
```

In [38]:
```
clf.tree_.compute_feature_importances()
```

Out[38]:
```
array([0.01333333, 0.01333333, 0.55072262, 0.42261071])
```

In [39]:
```
clf.feature_importances_
```

Out[39]:
```
array([0.01333333, 0.01333333, 0.55072262, 0.42261071])
```

- class_weight：样本类别权重设置

sklearn源码：

```
# compute the class weights for the entire dataset y
    if class_weight == "balanced":
        class_weight = compute_class_weight(class_weight,
                                            np.arange(len(self.classes_)),
                                                y)
        class_weight = dict(enumerate(class_weight))
```

```
elif class_weight == 'balanced':
    # Find the weight of each class as present in y.
    le = LabelEncoder()
    y_ind = le.fit_transform(y)
    if not all(np.in1d(classes, le.classes_)):

            raise ValueError("classes should have valid labels that are in y")
        recip_freq = len(y) / (len(le.classes_) *
                               np.bincount(y_ind).astype(np.float64))
    weight = recip_freq[le.transform(classes)]
```

compute_class_weight这个函数的作用是根据不平衡的输入样本的类别数量，为类别计算权重，从而平衡类别之间的权重。下面写段代码测试一下：

- （1）多分类问题

In [29]:
```
import sklearn.utils.class_weight as cw
import numpy as np

class_weight = 'balanced'
label = np.array([0]*9 + [1]*1 + [2]*2)
label
```

Out[29]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2])
```

In [39]:
```
classes = np.array([0,1,2])
weight = cw.compute_class_weight(class_weight='balanced',classes=classes,y=label)
weight
```

Out[39]:
```
array([0.44444444, 4.        , 2.        ])
```

In [31]:
```
weight[0] * 9
```

Out[31]:
```
4.0
```

In [22]:
```
weight[1] * 1
```

Out[22]:
```
4.0
```

In [23]:
```
weight[2] * 2
```

Out[23]:
```
4.0
```

In [54]:
```
# class_weight1 = {0:4/9,1:4,2:2}
class_weight1 = {0:1,1:3,2:5}
weight1 = cw.compute_class_weight(class_weight=class_weight1,classes=classes,y=label)
weight1
```

Out[54]:
```
array([1., 3., 5.])
```

In [55]:
```
np.bincount(label)*weight1
```

Out[55]:
```
array([ 9.,  3., 10.])
```

- （2）多标签问题

In [61]:
```
label_multi = np.array([[0,1,1],[1,0,1],[1,0,2],[1,2,1]])
label_multi
```

Out[61]:
```
array([[0, 1, 1],
       [1, 0, 1],
       [1, 0, 2],
       [1, 2, 1]])
```

In [84]:
```
#  For multi-output, the weights of each column of y will be multiplied.
class_weight2 = [{0:3,1:1},{0:1,1:2,2:2},{1:1,2:3}]
weight2 = cw.compute_sample_weight(class_weight=class_weight2,y=label_multi,indices=N
weight2 # 输出为每个样本的权重
```

Out[84]:
```
array([6., 1., 3., 2.])
```

对于第一个样本[0,1,1]的权重为每个列的标签对应权重的乘积。第一列标签0的设置权重为3，第二列标签1的设置权重为2，第三列标签1的设置权重为1，乘积为3$*$2$*$1=6。同理计算其余三个样本的权重乘积。

In [ ]: