2.2 决策树

有时是指学习方法，有时指学得的树

决策树是一类基于树结构进行决策的机器学习方法。决策树算法能够读取数据集合，并构建一棵用于分类的决策树。第 2.1 节介绍的 *k* 近邻算法可以完成很多分类任务，但其无法给出数据的内在含义，而决策树可以用于理解数据中蕴含的知识信息，具有使得数据形式非常易于理解的优势。因此，决策树算法可以针对不熟悉的数据集合，从中提取一系列规则，而构建出一棵决策树。决策树学习的目的是为了产生一棵泛化能力强，即处理未知示例能力强的决策树。

2.2.1 决策树的决策过程

决策树与人类在面临决策问题时构建的一种很自然的处理机制一致。对于一个邮件分类系统，需对一封邮件的属性进行智能分类，即判断一封邮件为“垃圾邮件”，还是“娱乐相关的邮件”，还是“工作相关的邮件”。面对这样的邮件分类问题，通常会进行一系列的判断。首先检测邮件的域名地址，如果邮件的域名地址为“myEntertainment.com”，则将其分类为“娱乐相关的邮件”。如果邮件的域名地址为其他，则进一步检测邮件中是否包含词语“工作”。如果邮件包含词语“工作”，则将其分类为“工作相关的邮件”。否则，将其分类为“垃圾邮件”。这个决策过程类似于 Python 编程中多重嵌套的“*if-elif-else*”的多分支选择结构，构建的决策树如图 2.2.1 所示。

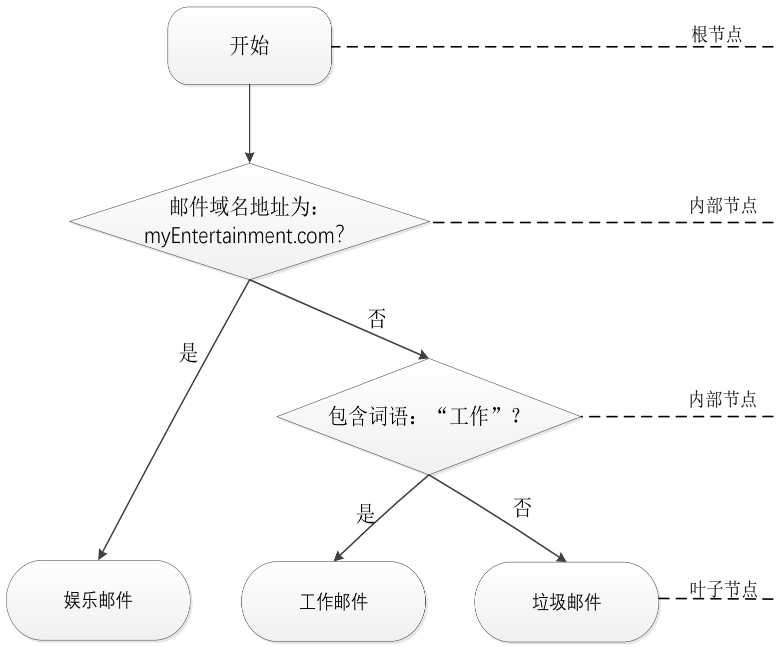

图 2.2.1 邮件分类系统的一棵决策树

显然，决策过程提出的每个判定问题都是对某个“属性”的测试。每个测试结果或是导出下个判定问题或是导出最终分类结果。下个判定问题和上次判定结果存在一定的关系，例

如：若邮件域名地址不为“myEntertainment.com”，再考虑邮件中是否包含词语“工作”，则仅考虑域名地址不为“myEntertainment.com”的邮件是否包含词语“工作”。一个“属性”被使用过后，则不再被使用。

从图 2.2.1 可看出，一棵决策树包含一个根结点、若干个内部结点和若干个叶子结点。叶子结点对应于判别结果。内部结点对应于属性测试。每个结点包含的样本集合根据属性测试的结果被划分到子结点中。根结点包含样本全集。根结点到每个叶结点的路径对应了一个测试判定序列。

2.2.2 决策树学习算法的基本流程

定义树函数 $CreateTree(Dataset, B)$，其中输入参数分别为数据集 $Dataset$ 和属性集合 B。首先，进行两次递归返回的判断。如果数据集中所有的类标签完全相同，则直接返回该类别标签。如果用完了属性集合中的所有属性，则返回数据集中样本出现类别最多的类别。其次，选取数据集中的最优判别属性，利用该最优属性创建树。一般采用 Python 语言里的字典变量来创建树，字典变量可存储树的所有信息，利于树形图的绘制。然后，遍历最优属性的所有属性值，在每个样本子集上递归调用函数 $CreateTree()$，创建分支结点。函数执行完毕时，会返回一个嵌套字典，字典的嵌套值代表着很多叶子结点的信息。如图 2.2.2 所示，为决策树学习算法的过程图。

输入： 训练集：$Dataset = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$；

属性集：$B = \{b_1, b_2, ..., b_d\}$

过程：定义树函数 $CreateTree(Dataset, B)$

if $Dataset$ 中样本全属于同一类别 C，***then***
　将叶结点类别标记为 C 类；***return***
endif
if $B = \varnothing$ ***then***
　将叶结点类别标记为 $Dataset$ 中样本数最多的类；***return***
endif
从 B 中选择最优划分属性 b_*
利用 b_* 创建树
for b_* 的每一个值 b_*^i do
　令 $Dataset_i$ 表示 $Dataset$ 中在 b_* 上取值为 b_*^i 的样本子集
　if $Dataset_i$ 为空，*then*
　　将分支结点标记为叶结点，其类别标记为 $Dataset$ 中样本最多的类;return
　else
　　以 $CreateTree(Dataset_i, B \setminus \{b_*\})$ 创建分支结点
　end if
end for
输出：一棵决策树

图 2.2.2 决策树学习基本算法

显然，决策树的生成是一个递归过程。有 3 种情形会导致递归返回：

(1)当前结点包含的样本全属于同一类别，无需划分；

(2)当前属性集为空，无法划分，将当前结点标为叶结点，类别设定为该结点所含样本最多的类别;

(3)当前结点包含的样本集合为空，不能划分，将当前结点标为叶结点，类别设定为其父结点所含样本最多的类别。

2.2.3 划分属性的选择

由决策树学习的基本算法可知，决策树学习的关键之一为“如何选取最优划分属性”。划分数据集的大原则是将无序的数据变得更加有序。具体而言，随着数据集划分过程的进行，希望决策树的分支结点所包含的样本尽可能属于同一类别，即结点的纯度越来越高。

信息论是量化处理信息的分支学科，可以在划分数据前后利用信息论来度量数据包含的信息特征。一般采用信息熵(Information Entropy) 来度量样本集合中包含的信息量。

信息熵是度量样本集合纯度的一种指标。假设当前样本集合 $Dataset$ 中第k类样本所占的比例为 p_k ($k=1,2,...,K$)，则 $Dataset$ 的信息熵为：

$$Ent(Dataset) = -\sum_{k=1}^{K} p_k \log_2 p_k \qquad (2.2.1)$$

$Ent(Dataset)$ 表示所有类别样本包含的信息大小的期望值， $Ent(Dataset)$ 值越小，则 $Dataset$ 的纯度越高。

假定离散属性 b 有 d 个可能的取值 $\{b_1, b_2, ..., b_d\}$ ，若使用 b 对 $Dataset$ 进行划分，则会产生 V 个分支结点。其中，第 v 个结点包含了 $Dataset$ 中所有在属性 b 取值为 b^v 的样本，记为 $Dataset_v$ 。根据式(2.2.1)算出 $Dataset_v$ 的信息熵。不同分支结点包含的样本数不同，给分支结点赋予权重 $|Dataset_v|/|Dataset|$ ，即样本数越多，则分支结点权重越大。因此，若用属性 b 对 $Dataset$ 进行划分，其信息增益为：

$$Gain(Dataset, b) = Ent(Dataset) - \sum_{v=1}^{V} \frac{|Dataset_v|}{|Dataset|} Ent(Dataset_v) \qquad (2.2.2)$$

显然，若信息增益越大，则用属性 b 对 $Dataset$ 进行划分对应的分支结点的纯度越高，即分支结点包含的样本尽可能属于同一类别。因此，可利用信息增益作为指标，选择决策树的划分属性，即最优划分属性为：

$$b_* = \underset{b}{argmax} Gain(Dataset, b) \qquad (2.2.3)$$

以表 2.2.1 的西瓜数据集 2.0 为例，数据集包含17 个训练样本，用来学习一棵决策树，以预测没切开的西瓜是否好瓜。

决策树开始学习时，根结点包含 $Dataset$ 中的所有样本。正样本占 $p_1 = 8/17$ ，负样本占 $p_2 = 9/17$ 。则根结点的信息熵为：

$$Ent(Dataset) = -\sum_{k=1}^{2} p_k \, log_2 \, p_k = -\left(\frac{8}{17} log_2 \frac{8}{17} + \frac{9}{17} log_2 \frac{9}{17}\right) = 0.998 \qquad (2.2.4)$$

编号	色泽	根蒂	敲声	纹理	脐部	触感	好瓜
1	青绿	蜷缩	浊响	清晰	凹陷	硬滑	是
2	乌黑	蜷缩	沉闷	清晰	凹陷	硬滑	是
3	乌黑	蜷缩	浊响	清晰	凹陷	硬滑	是
4	青绿	蜷缩	沉闷	清晰	凹陷	硬滑	是
5	浅白	蜷缩	浊响	清晰	凹陷	硬滑	是
6	青绿	稍蜷	浊响	清晰	稍凹	软粘	是
7	乌黑	稍蜷	浊响	稍糊	稍凹	软粘	是
8	乌黑	稍蜷	浊响	清晰	稍凹	硬滑	是
9	乌黑	稍蜷	沉闷	稍糊	稍凹	硬滑	否
10	青绿	硬挺	清脆	清晰	平坦	软粘	否
11	浅白	硬挺	清脆	模糊	平坦	硬滑	否
12	浅白	蜷缩	浊响	模糊	平坦	软粘	否
13	青绿	稍蜷	浊响	稍糊	凹陷	硬滑	否
14	浅白	稍蜷	沉闷	稍糊	凹陷	硬滑	否
15	乌黑	稍蜷	浊响	清晰	稍凹	软粘	否
16	浅白	蜷缩	浊响	模糊	平坦	硬滑	否
17	青绿	蜷缩	沉闷	稍糊	稍凹	硬滑	否

表 2.2.1 西瓜数据集 2.0

然后，计算利用“每个属性”进行数据划分的信息增益。以属性“纹理”为例，其有 3 个可能的取值：{清晰，稍糊，模糊}。用“纹理”对 *Dataset* 划分，可得 3 个子集，即 $Dataset_1:(纹理=清晰)$、$Dataset_2:(纹理=稍糊)$、$Dataset_3:(纹理=模糊)$。分别计算 $Dataset_1$、$Dataset_2$、$Dataset_3$ 中正样本类别及负样本类别所占的比例，代入式(2.2.2)，得到利用“纹理”对 *Dataset* 划分的信息增益：

$$Gain(Dataset, 纹理) = Ent(Dataset) - \sum_{v=1}^{3} \frac{|Dataset_v|}{|Dataset|} Ent(Dataset_v)$$

$$= 0.381 \tag{2.2.5}$$

经比较，属性“纹理”带来的信息增益最大，于是首先选“纹理”为最优划分属性，对根结点进行划分，如图 2.2.3 所示。显然，当前属性的取值个数即为当前结点的分支数。

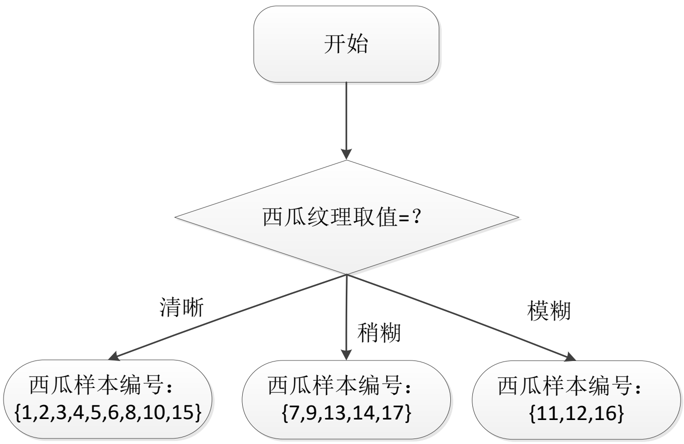

图 2.2.3 基于”纹理”对根结点划分

然后，决策树学习算法对每个分支结点做进一步划分。以第 1 个分支结点

$\{''\text{西瓜纹理取值}=\text{清晰}''\}$ 为例， $Dataset_1$ 中包含的样本数为 9 个，即西瓜样本编号：$\{1,2,3,4,5,6,8,10,15\}$，此时其可用属性集合为{色泽，根蒂，敲声，脐部，触感}，已排除掉“纹理”属性。基于 $Dataset_1$ 计算各属性的信息增益，经比较，“根蒂”，“脐部”，“触感”三个属性均取得了最大的信息增益，因此，可任选其中之一作为划分属性。类似的，对每个分支结点进行上述操作，直到得到最终的决策树。

2.2.4 其他属性选取指标

（1）增益率

在上面决策树的生成过程中，若将表 2.2.1 中“编号”也作为一个候选划分属性，则算出其信息增益为 0.998，远大于其他候选信息增益。然而，“编号”属性的取值个数为 17，利用其对根结点进行划分，将产生 17 个分支，每个分支结点仅包含 1 个样本。这些分支结点的纯度已达最大，即分支结点包含的样本仅占据 1 个类别。但这样的决策树不具有泛化能力，不具有实际应用价值。

显然，根据最大化信息增益的准则，其倾向于选取可取值数目较多的属性作为最优划分属性，为降低这种偏向带来的不良影响，著名的 C4.5 决策树算法不直接使用信息增益，而使用增益率来选择最优划分属性。增益率定义为：

$$Gain\_ratio(Dataset,b)=\frac{Gain(Dataset,b)}{IV(b)} \tag{2.2.6}$$

其中，

$$IV(b)=-\sum_{v=1}^{V}\frac{|Dataset_v|}{|Dataset|}log_2\frac{|Dataset_v|}{|Dataset|} \tag{2.2.7}$$

$IV(b)$ 为属性 b 的固有值，属性 b 的可能取值数目越多，则 $IV(b)$ 越大。例如，对西瓜数据集 2.0，有 $IV(\text{触感})=0.874$，对应 $V=2$；$IV(\text{色泽})=1.580$，对应 $V=3$；$IV(\text{编号})=4.088$，对应 $V=17$。

因此，增益率准则倾向于选取可取值数目较少的属性作为最优划分属性，C4.5 算法并不是直接选择增益率最大的候选划分属性，而是先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择增益率最高的。

（2）基尼系数

由 Breiman 在 1984 年提出的 CART(Classification and Regression Tree)决策树，其使用“基尼系数”（Gini Index）来划分属性。基尼系数是指国际上通用的、用以衡量一个国家或地区居民收入差距的常用指标。基尼指数最早由意大利统计与社会学家 Corrado Gini 在 1912 年提出。基尼系数最大取值为“1”，最小取值为“0”。基尼系数越接近 0，表明收入分配越趋向平等。国际惯例把基尼系数取值为 0.2 以下视为收入绝对平均；基尼系数取值为 $0.2-0.3$ 视为收入比较平均；基尼系数取值为 $0.3-0.4$ 视为收入相对合理；基尼系数取值为 $0.4-0.5$ 视为收入差距较大；当基尼系数达到 0.5 以上时，则表示收入悬殊。显然，对于经济体而言，基尼系数取值越小越好。

基尼系数的计算方式如下：

$$Giniindex(Dataset,b)=\sum_{v=1}^{V}\frac{|Dataset_v|}{|Dataset|}Gini(Dataset_v,b) \tag{2.2.8}$$

$$Gini(Dataset_v,b)=1-\sum_{k=1}^{K}p_k^2 \tag{2.2.9}$$

若使用b对$Dataset$进行划分，第v个结点包含了$Dataset$中所有在属性b取值为b^v的样本，记为$Dataset_v$。其中p_k表示利用属性b对$Dataset$进行划分时，$Dataset_v$中每个类别的样本数占$Dataset_v$中总样本数的比例。$\frac{|Dataset_v|}{|Dataset|}$为两个集合$Dataset_v$与$Dataset$中样本数的比值。

以表 2.2.2 的工作数据集为例，数据集包含8个训练样本，3个属性分别为{工资、压力、平台}。

编号	工资	压力	平台	工作
1	1	1	2	好
2	0	1	0	好
3	1	0	0	好
4	0	1	0	好
5	0	1	1	不好
6	1	1	1	好
7	0	0	2	不好
8	0	0	1	不好

表 2.2.2 工作数据集

首先，“工资”属性有两个取值，分别为0和1。当“工资$=1$”时，样本数为3。同时，这3个样本中，工作的类别都为“好”。因此，

$$Gini(Dataset_1, 工资=1) = 1-\left(\frac{3}{3}\right)^2-\left(\frac{0}{3}\right)^2=0 \tag{2.2.10}$$

当“工资$=0$”时，样本数为5。同时，这5个样本中，工作类别为“好”的样本数为2，工作类别为“不好”的样本数为3。因此，

$$Gini(Dataset_0, 工资=0) = 1-\left(\frac{2}{5}\right)^2-\left(\frac{3}{5}\right)^2=\frac{12}{25} \tag{2.2.11}$$

则“工资”属性的基尼系数为

$$Giniindex(Dataset, 工资)=\frac{3}{8}\times\left[1-\left(\frac{3}{3}\right)^2-\left(\frac{0}{3}\right)^2\right]+\frac{5}{8}\times\left[1-\left(\frac{2}{5}\right)^2-\left(\frac{3}{5}\right)^2\right]=0.3 \tag{2.2.12}$$

同理，计算可得“压力”、“平台”的基尼系数。经比较，“工资”属性的基尼系数最小。根据基尼系数最小准则，优先选择“工资”属性作为$Dataset$的第一划分属性。

2.2.5 剪枝处理

剪枝是决策树学习算法对付“过拟合”的主要手段。在决策树学习过程中，为了正确分类训练样本，结点划分过程将不断重复，可能会造成分支过多。这时可能因为把训练样本学得太好，以致于把训练集自身的一些特点（比如噪声）当作所有数据都具有的一般性质而导致过拟合。因此，可通过主动去掉一些分支来降低过拟合的风险。

决策树剪枝的基本策略有“预剪枝”和“后剪枝”。所谓预剪枝，即在决策树生成过程中，对每个结点划分前先进行估计，若当前结点的划分不能带来决策树泛化性能的提升，则停止划分，并将当前结点标记为叶结点。后剪枝的定义为：先从训练集生成一棵完整的决策树，然后自底向上对非叶结点进行考察，若将该结点对应的子树替换为叶结点能带来决策树泛化性能的提升，则将该子树替换为叶结点。

2.2.6 决策树的核心代码实现

本节将详细介绍决策树基本算法的 Python 代码实现，利用决策树实现简单鱼类别鉴定

任务，并给出代码的简单运行实例。相信读者在此基础上能够在实验课中进一步完善和改进代码，完成更具挑战性和实用性的应用任务。

“简单鱼数据集”如表 2.2.3 所示，表中包含 5 个海洋生物样本，每个样本有两个特征，即“不浮出水面是否能生存”和“是否有脚蹼”。标签为“属于鱼类”和“不属于鱼类”，分别用“是”和“否”表示。

	不浮出水面是否能生存	是否有脚蹼	属于鱼类
1	是	是	是
2	是	是	是
3	是	否	否
4	否	是	否
5	否	是	否

表 2.2.3 简单鱼数据集

首先来看数据集创建函数 createDataSet()：

```
def createDataSet():
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    labels = ['no surfacing','flippers']
    return dataSet, labels
```

该函数 createDataSet()创建了“简单鱼数据集”，1 行为 1 个样本，共有 5 个样本；每行有 3 列，前 2 列对应 2 个属性值（取值为‘0’或‘1’），第 3 列对应类别标签（取值为‘yes’或‘no’）。labels 为属性名称列表，共有两个元素'no surfacing'和'flippers'，分别表示样本“不浮出水面是否能生存”及样本“是否有脚蹼”。'no surfacing'取值为 1 表示样本“不浮出水面能生存”，反之“不能生存”；'flippers'取值为 1 表示样本“有脚蹼”，反之“不具有脚蹼”。类别标签取值为‘是’，表明该样本为鱼类；类别标签取值为‘否’，表明该样本不是鱼类。

接下来定义香农熵计算函数 calcShannonEnt()（对应式（2.2.1））：

```
def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet: #用字典统计样本不同类别出现次数
        currentLabel = featVec[-1]
        if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt -= prob * log(prob,2)
    return shannonEnt
```

该函数首先计算数据集 dataSet 中样本的总数。然后，创建一个数据字典 labelCounts，它的“键”是特征向量 featVec 的最后一列，即类别标签。如果当前“键”不存在，则扩展

字典，并将当前“键”加入字典。每个“键”都记录了当前类别出现的次数。最后，使用所有类别标签出现的频率作为类别标签出现的概率，并利用此概率计算香农熵。

接下来，定义根据指定的属性进行数据集划分的函数 splitDataSet()：

```
def splitDataSet(dataSet, axis, value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]        #去掉用于划分的属性
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet
```

该函数根据指定的属性进行数据集划分，其输入参数 dataSet 为由列表元素组成的列表，axis 为划分数据集的属性索引号，value 为属性取值。在函数内部语句的第一行声明一个新列表对象 retDataSet，原因在于该函数 splitDataSet()在同一数据集上被调用多次，为了不修改原始数据集，创建此新列表对象。数据集 dataSet 的各个元素也为列表，因此，采用 for 循环遍历数据集中的每条数据，一旦样本的属性值和 value 相等，则将该样本添加到新列表 retDataSet 中，同时必须删掉该属性值，以免该属性值在决策树的创建过程中被重复使用。该段代码的功能为：当按照某个属性划分数据集时，需要将所有符合要求的样本抽取出来，并删掉样本的该属性值。

接下来，定义最优划分属性选择函数 chooseBestFeatureToSplit()（对应式(2.2.3)）：

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1          #最后一列是类别标签
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):              #遍历所有样本属性
        featList = [example[i] for example in dataSet] #得到所有样本的第 i 个属性
        uniqueVals = set(featList)            #得到唯一属性构成的集合
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy         #计算信息增益，即熵的减少量
        if (infoGain > bestInfoGain):           #与当前最大增益进行比较
            bestInfoGain = infoGain             #得到当前最大增益
            bestFeature = i
    return bestFeature                          #返回最大增益对应的属性编号
```

该函数实现选取最优划分特征，并划分数据集，其输入参数 dataSet 为由列表元素组成的列表，每个列表元素代表一个样本，显然，所有的列表元素都须具有相同的数据长度。同时，每个列表元素的最后一列为当前样本的类别标签。因此，函数的第一行可计算出数据集包含的属性特征个数。

函数的第二行代码计算了整个数据集的原始香农熵，即最初的数据集的纯度值，用于和划分后的数据集的熵值进行比较。最外层的 for 循环遍历了数据集的所有属性，for 循环内的第一条语句使用列表推导式创建了新的列表，即将数据集的第 i 个属性写入这个新列表

featList 中。然后，使用集合（set）类型去除 featList 中的重复值，存入集合变量 uniqueVals 中。

接着，内层的 for 循环用以遍历 uniqueVals 中第 i 个特征的取值，通过函数 splitDataSet() 利用第 i 个属性值对数据集进行划分，计算划分后数据集的新熵值，并对每个属性值划分的数据集的熵值求和。然后计算划分数据前后信息增益的大小，即数据集无序程度的减少。最后，比较所有属性带来的信息增益的大小，返回最大信息增益值对应的属性索引号。

接下来，定义最大类别选择函数 majorityCnt()：

```
def majorityCnt(classList):
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
```

该函数用以计算出现次数最多的类别标签。输入参数 classList 为列表变量，其列表元素为数据集的类别标签。首先，创建数据字典 classCount，用以存放每个类别标签对应的样本个数。然后，利用 sorted()对每个类别标签的样本个数按照由大到小进行排序，并返回样本个数最多的类别标签。

接下来，定义决策树创建函数 createTree()：

```
def createTree(dataSet, labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]     #当样本都属于同一个类别就停止划分
    if len(dataSet[0]) == 1:     #当属性都用完也要停止划分
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]        #生成一个副本，避免 labels 被修改
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),
subLabels)
    return myTree
```

该函数 createTree()用以创建决策树，其输入参数为数据集 dataSet 和属性名称列表 labels。第一行代码创建了列表变量 classList，存放数据集的类别标签。接下来第一条 if 语句用以判断数据集的长度是否等于类别标签列表中第 0 个元素的个数，若相等，则说明数据集的所有样本都属于同一种类别，则直接返回该类别标签，无须继续判断。第二条 if 语句用以判断第 0 条样本的长度是否为 1，若是，则说明所有的属性已用完，直接返回出现次数最多的类别值。

接下来，调用函数 chooseBestFeatureToSplit()选择最佳的划分属性，即 bestFeat 和其对应的属性名称 bestFeatLabel。接着，开始创建树 myTree，其类型为嵌套的字典结构，初始

“键”为最佳划分属性 bestFeatLabel，即以该最佳划分属性为根结点开始创建树。为了避免该最佳划分属性被重复使用，接下来就将其从属性名称列表 labels 中删除掉。紧接着，通过 for 循环对最佳划分属性的取值进行遍历。for 循环的第一行语句“subLabels = labels[:]”的作用是复制了属性名称列表 labels，将其存储在新列表变量 subLabels 中。原因在于 python 语言中当函数参数是列表类型时，参数是按照引用方式传递的（即传址），为了保证每次调用函数 createTree()时不改变原始列表的值，使用新变量 subLabels 代替原始列表。

然后，利用函数 splitDataSet()获取最佳划分属性划分的子数据集，并将子数据集及对应的属性名称列表 subLabels 传入函数 createTree()，递归调用函数 createTree()，并将返回值插入到字典变量 myTree 中。因此，函数执行结束时，字典 myTree 中将会嵌套很多代表着树结点信息的字典结构，一棵完整的决策树也就创建完成。一棵完整的决策树实际就是一个嵌套字典结构。

依靠训练数据构造了决策树之后，就可将其用于实际数据的分类。决策树分类函数 classify()采用递归方式完成这一过程：

```
def classify(inputTree, featLabels, testVec):
    firstStr = list(inputTree.keys())[0]
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)
    key = testVec[featIndex]
    valueOfFeat = secondDict[key]
    if isinstance(valueOfFeat, dict):
        classLabel = classify(valueOfFeat, featLabels, testVec)
    else: classLabel = valueOfFeat
    return classLabel
```

该函数输入参数分别为决策树 inputTree、属性名称列表 featLabels、待分类样本的属性取值列表 testVec。首先，通过取决策树 inputTree 的“键”列表的第 0 个元素，获得最佳划分属性 firstStr，接着得到决策树以最佳划分属性为“键”的值 secondDict，其为嵌套字典结构。然后，通过 index 方法获取最佳划分属性的索引值 featIndex，并据此得到待分类样本的对应属性名称 key。然后得到嵌套字典 secondDict 以 key 为“键”对应的值 secondDict[key]。如果该值为字典结构，说明仍需继续进行类别判断，因此递归调用函数 classify()，此时 classify()的输入参数为 secondDict[key]，属性名称列表 featLabels，及待分类样本的属性取值列表 testVec。如果该值不是字典，说明已经到达叶子结点，则直接返回当前结点的类别标签即可。

为了更直观地查看决策树的分类结果，可以将决策树可视化出来。Python 并没有提供绘制树的工具，因此需利用 Matplotlib 库绘制树形图。为了绘制这棵完整的树，必须知道叶子结点的个数，以确定 x 轴的长度。还需要知道树的层树，以确定 y 轴的高度。因此定义两个函数 getNumLeafs()和 getTreeDepth()，分别获取叶子结点的数目和树的层数，如下：

```
def getNumLeafs(myTree):
    numLeafs = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict': #如果该结点是字典，就不是叶结点
            numLeafs += getNumLeafs(secondDict[key])
        else:     numLeafs +=1
```

```
    return numLeafs

def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict': #如果该结点是字典，就不是叶结点
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:     thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth
    return maxDepth
```

函数 getNumLeafs()和 getTreeDepth()具有相同的结构。firstStr 是树的根结点，从其出发得到下一个结点 secondDict。然后利用 type()函数判断其是否为叶结点，如果不是就递归调用。如果是，就不再递归调用。特别注意两个函数的不同之处，getNumLeafs()函数遍历整棵树，累计叶子结点的个数，并返回该值即可。而 getTreeDepth()函数还需要比较不同分支的层数，以得到最大层数 maxDepth。

定义函数 retrieveTree()，用来存储构造好的树。以“简单鱼数据集”为例：

```
def retrieveTree(i):
    listOfTrees =[{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}},
                  {'no surfacing': {0: 'no', 1: {'flippers': {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}
                   ]
    return listOfTrees[i]
```

将以上代码添加至文件 trees.py，来验证一下代码的正确性：

```
>>> import trees
>>> myDat, labels=trees.createDataSet()
>>> labels
[‘no surfacing’,’flippers’]
>>> myTree=trees.retrieveTree(0)
>>> myTree
{‘no surfacing’:{0:’no’,1:{flippers:{0:’no’,1:’yes’}}}}
>>> trees.classify(myTree, labels, [1,0])
‘no’
>>> trees.classify(myTree,labels,[1,1])
‘yes’
>>> trees.getNumLeafs(myTree)
3
>>> trees.getTreeDepth(myTree)
2
```

代码运行正常，也能得到正确的结果。

接下来定义函数 createPlot()，完成决策树的可视化。其创建绘图区，计算决策树的全局尺寸，并调用递归绘图函数 plotTree()。plotTree.totalW 和 plotTree.totalD 为全局变量，分别存储树的宽度和深度，这两个变量用于将树绘制在水平方向和垂直方向的中心位置。全局变量 plotTree.xOff 和 plotTree.yOff 用于追踪已经绘制的结点位置以及放置下一个结点的恰当

位置。plotTree()调用 plotMidText()函数计算父结点和子结点的中间位置，并在此处添加简单的文本标签信息，然后调用 plotNode()画出结点。注意，画完后需要按比例减少垂直位置 plotTree.yOff，因为是自顶向下绘制图形。plotTree()递归调用的规则同样是“判断是否叶子结点”，如果是则在图形上画出叶子结点，如果不是则递归调用 plotTree()函数。类似地，每次画完叶子结点，需要向右调整水平位置，再画下一个叶子结点。还要注意，画完子树的所有叶子结点后，垂直位置应按比例向上调整。

```
def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    createPlot.ax1 = plt.subplot(111, frameon=False, **axprops)    #不带刻度
    #createPlot.ax1 = plt.subplot(111, frameon=False) #带刻度，方便演示
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()

def plotTree(myTree, parentPt, nodeTxt):
    numLeafs = getNumLeafs(myTree)   #叶结点数决定树的宽度
    depth = getTreeDepth(myTree)
    firstStr = list(myTree.keys())[0]       #得到根结点
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict': #如果该结点是字典，就不是叶结点
            plotTree(secondDict[key],cntrPt,str(key))          #递归调用
        else:    #是叶结点就将其打印出来
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD

def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
    #createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)
    createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=0)

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy=parentPt,   xycoords='axes fraction',
```

```
xytext=centerPt, textcoords='axes fraction',
va="center", ha="center", bbox=nodeType, arrowprops=arrow_args )
```

将以上代码添加至文件 trees.py，来验证一下代码的正确性：

```
>>> import trees
>>> myTree=trees.retrieveTree(0)
>>> trees.createPlot(myTree)
```

绘制结果如图 2.2.4 所示，则一棵完整的树绘制成功。

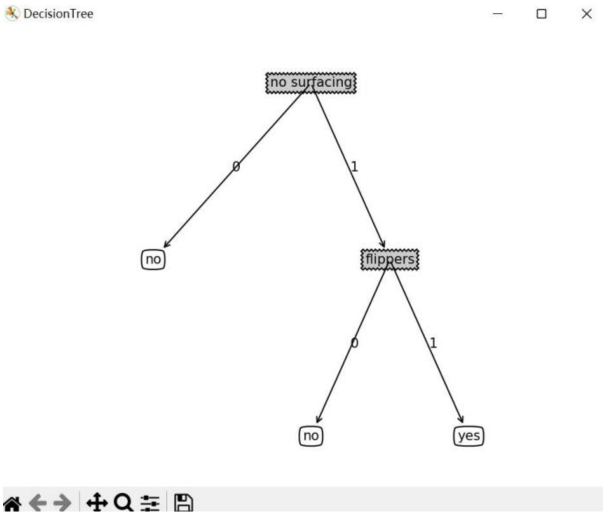

图 2.2.4 决策树绘制结果

按照如下命令变更字典，重新绘制树形图，如图 2.2.5 所示，为一个三分支的树形图。

```
>>> myTree['no surfacing'][3]='maybe'
>>> myTree
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}, 3: 'maybe'}}
>>> trees.createPlot(myTree)
```

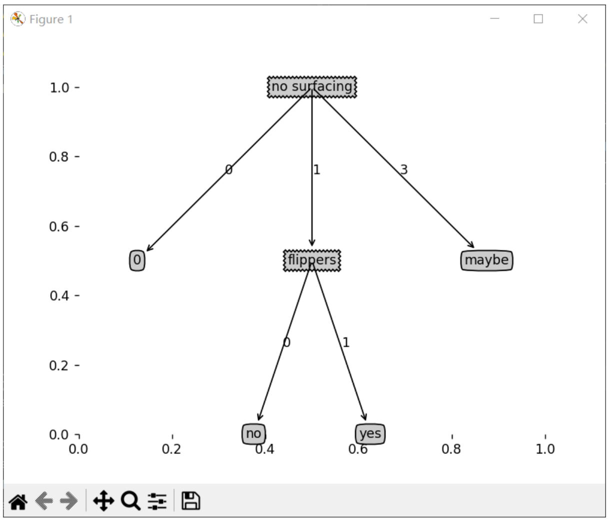

图 2.2.5 超过两个分支的树形图

习题 2.2

1. 解释图 2.2.1 中根结点、内部结点及叶子结点的含义。
2. 请给出式(2.2.5)的详细计算过程，验证其正确性。
3. 针对表 2.2.1 中的西瓜数据集 2.0，根据式(2.2.3)分别计算利用属性“色泽”、“根蒂”、“脐部”进行数据划分的信息增益。
4. 解释信息增益准则的含义。
5. 解释决策树算法学习过程中产生过拟合现象的原因，如何应对此过拟合问题？
6. 在 Python 中树是以什么数据类型存储的？