第3章 决 策 树

本章学习目标

- 理解决策树的构建；
- 掌握决策树的实现方法；
- 掌握树形图的绘制方法；
- 了解决策树的剪枝技术。

决策树算法是一种非参数的监督学习方法，常被用于从一组无序、无规则的样本数据中推理出决策树表示形式的分类规则，例如数据挖掘任务。顾名思义，决策树以树形数据结构来进行分类或预测决策，在分类应用中，决策树中的每个结点构成类标签，叶子结点是最终的分类标号；树中的分支由决策规则组成。决策树是最常见的一种机器学习算法，它易于实现，可解释性强，符合人类的直观思维。本章将对决策树的构建以及实现方法进行讲解。

3.1 决策树与信息熵

3.1.1 决策树简介

决策树通常用一棵倒置的树结构来表示数据间的逻辑关系，基于数据的特征进行判断，进而得到分类或回归结果。树结构中通常包含三种结点，分别为根结点、子结点和叶子结点。根结点是树的最顶端的结点，每棵决策树只会有一个根结点；子结点对应于每一个分裂问题，该结点的每一个后继分支对应于该特征的一个可能值；叶子结点是带有分类标签的数据集合，即样本所属的分类。当决策树不断分裂直到无法再分出子结点时称该结点为叶子结点。接下来，通过一个简单示例展示决策树模型。

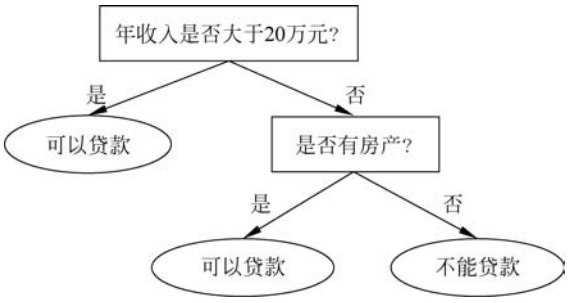


图 3.1 判断用户是否能贷款

假设银行要使用机器学习算法来确定是否给客户发放贷款，为此需要考察客户的年收入以及是否有房产这两个指标。判断是否给用户贷款的过程很简单，如图 3.1 所示。

图 3.1 是一个典型的决策树，图中的矩形表示子结点，即判断模块(decision block)；椭圆表示叶子结点，即终止模块(terminating block)，对应

于通过决策树算法分析出的各种结论；箭头表示分支(branch)，即决策规则。决策过程从树的根结点开始，在子结点处进行判断，直到到达一个叶子结点处，得到决策结果。决策树由一系列分层嵌套的判定规则组成，是一个递归的结构。

第 2 章介绍了 K 近邻算法，其最大的缺点是无法给出数据的内在含义。而在决策树中，树形结构使得数据形式非常容易理解，这是决策树的最大优势。在此总结出决策树算法的三个主要特点，具体如下所示。

- 决策树算法适用于数值型和标称型数据，通过决策树可以读取数据集合，提取数据中蕴含的规则。
- 决策树算法在解决分类问题时，具有较低的复杂度、便用性和高效性。
- 决策树算法可以用于处理具有不相关特征的数据，它的树结构可以很容易地构建易于理解的规则。

决策树算法的缺点在于难以处理数据缺失的情况，容易出现过拟合，并且树结构本身难以突出数据集中特征之间的相关性。

决策树模型的学习过程主要有以下三个步骤。

(1) 特征选取。特征选取是指从大量训练数据特征中提取一个特征值作为当前结点的分裂标准。特征值的选择标准有着许多不同量化评估标准，也因此衍生出不同的决策树算法。

(2) 决策树生成。根据上一步所选择的特征评估标准，从上至下递归地生成子结点，直到数据集不再可分裂，此时停止决策树的生长。

(3) 剪枝。决策树容易出现过拟合问题，需要通过剪枝来缩小树的结构规模，从而缓解过拟合问题。常见的剪枝技术有预剪枝技术和后剪枝技术。

在构建决策树时，首先要解决的问题是判断出当前数据集中哪个特征在划分数据分类时起决定性作用。通过对每个特征进行评估来找到决定性的特征。在选取好待划分的特征后，原始数据集将根据这个特征被划分为若干个数据子集。该数据子集会分布在第一个决策点的所有分支上。如果某个分支下的数据属于同一类型，则无须进一步对数据集进行分割。如果数据子集内的数据不属于同一类型，则需要递归地重复划分数据子集的过程，直到每个数据子集内的数据类型相同。

决策树的训练流程遵循简单且直观的“分而治之”策略，接下来通过一段伪代码来演示创建分支的过程：

```
检测数据集中的每个子项是否属于同一类型:
If Yes return 类标签
Else
    找出用于划分数据集的最优特征
    进行数据集的划分
    构建新的分支结点
    for 每个被划分的子集:
            递归调用本算法并添加返回结果到分支结点中
    return 分支结点
```

上述伪代码通过递归的方式实现了分支的构建。本章后续内容会演示通过 Python 实

现该过程的方法。

常用的 5 种决策树算法如下所示。

- D3 算法，其核心是在决策树的各级结点上，使用信息增益方法作为特征的选择标准来确定生成每个结点时所选取的最优特征。该算法的缺点是只适用于离散型特征描述。
- C4.5 算法，对 ID3 算法进行了改进，使用信息增益率来选择结点特征。C4.5 算法克服了 ID3 算法的不足，既可以处理离散型特征描述任务，也可以处理连续型特征描述任务。
- CART 算法，一种十分有效的非参数分类和回归方法，通过构建树、剪枝和评估来构建二叉树结构。当叶子结点是连续变量时，该树为回归树；当叶子结点是分类变量时，该树为分类树。
- SLIQ 算法，对 C4.5 算法进行了改进，在决策树的构建过程中采用了“预排序”和“广度优先策略”两种技术。在建树阶段，SLIQ 算法会针对连续属性采取预先排序技术与广度优先相结合的策略生成树，对离散属性采取快速求子集算法确定划分条件。SLIQ 算法具有可伸缩性良好、学习时间短、能处理常驻磁盘的数据集以及处理结果准确的优点。
- SPRINT 算法，一种可扩展的、可并行的归纳决策树，它完全不受内存限制，运行速度快，且允许多个处理器协同创建一个决策树模型。SPRINT 算法对 SLIQ 算法进行了进一步的改进，去掉了 SLIQ 中需要驻留于内存的类别列表，将类别列合并到了每个特征列表中。SPRINT 算法的优点是在寻找每个结点的最优划分特征时变得更简单。其缺点是对非分裂属性的属性列表进行划分非常困难。

3.1.2 信息与自信息

在进一步学习决策树算法之前，有必要掌握一些必要的基础概念，如信息与自信息的概念。

广义上的信息是指事物运动时发出的信号所带来的消息，是事物存在方式和运动规律的一种表现形式。不同的事物具有不同的存在方式和运动规律，从而构成了各种事物的不同特征。信息普遍存在于自然界、社会界以及人的思维之中，是客观事物千差万别的本质特征的反映。信息分为两大类：自然信息与社会信息。

消息是指信息的具体反映形式，是信息的实质内容。不同的消息中所包含的信息量是不同的。只有被消息的接收者了解并认识的内容(这部分内容接收者事先不知道)才蕴含着信息。

自信息(self-information)是信息的度量单位，由克劳德·香农提出，用来衡量单一事件发生时所包含的信息量多寡，它的单位是 b(或者 nats)。所接收到的自信息的量与具体发生的事件有关，自信息的大小与随机事件的概率有关。概率越小的事件发生后所包含的自信息越多，概率越大的事件发生后所包含的自信息越少。

信息论的基本观点是：“一个极小概率事件的发生，比一个大概率事件的发生所提供信息更多。”例如，学校给学生发放了一则通知：“周一早上照常上课。”很显然，学校周一上课属于十分正常的事件，在没有其他外在环境干涉的情况下，这条通知含有的自信息是极少

的。如果学校将通知改成“周一早上放假”,那么这条通知的自信息便多于上一条通知。这是因为相较于“周一上课”,“周一放假”属于小概率事件。

通过信息论的思想来量化信息,需要注意以下几点:

- 概率越大的事件发生后所产生的自信息越少。在极端情况下,必定发生事件的自信息含量为 0。
- 概率越小的事件发生后所包含的自信息越多。
- 独立事件应具有增量的信息。例如,投掷硬币 2 次,结果都是正面朝上所传递的自信息是只出现 1 次正面朝上的自信息含量的 2 倍。

3.1.3 信息熵

一般情况下,数据集划分的大原则是将原本无序的数据整理成更加有序的分组。其中一种对杂乱无章的数据进行整理的方法便是通过信息论度量信息。信息论属于量化处理信息的分支科学。

如果待分类的数据集 D 中第 i 类样本所占的比例为 $p(x_i),(i=1,2,\cdots,n)$,n 表示分类数量,则类别 x_i 的自信息表达式如下所示。

$$I(x_i)=-\mathrm{lb}p(x_i)$$

为了计算熵,需要首先通过下列公式来计算所有类别的可能值所包含的信息期望值表达式。

$$\mathrm{Ent}(D)=-\sum_{i=1}^{n}p(x_i)\mathrm{lb}p(x_i)$$

本章后续小节将通过实例来演示决策树进行分类的原理和代码实现方法。

假设有如表 3.1 所示的 5 种海洋生物,其中包含了它们的两种特征:①是否可以始终保持在水下生存;②是否具有脚蹼。表 3.1 还对这 5 种海洋生物进行了划分,表明它们是否属于鱼类。现在需要确定依据两种特征中的哪一种进行划分,才可以更好地对这 5 种海洋生物是否属于鱼类进行准确的判断。

表 3.1 5 种海洋生物的数据

序　　号	是否可以始终保持在水下生存	是否具有脚蹼	是否属于鱼类
1	是	是	是
2	是	是	是
3	是	否	否
4	否	是	否
5	否	是	否

首先,需要计算数据集的信息熵,具体方法如例 3.1 所示。

【例 3.1】 计算数据集的信息熵。

```
1   from math import log
2   import operator
3   def calcShannonEnt(dataSet):
4     numEntries = len(dataSet)          #声明数据集中样本总数
5     labelCounts = {}                   #创建字典
```

```
    for featVec in dataSet:                    #所有可能分类的数量和发生频率
        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) #log base 2
    return shannonEnt
```

上述代码中所创建的字典的键值为最后一列的数值。当前键值不存在时，则扩展字典并将当前键值加入字典。每个键值都记录了当前类别出现的次数。最后，使用所有类标签的发生频率计算类别出现的概率，通过该概率计算信息熵，统计所有类标签发生的次数。代码中的 import operator 是为后续递归构建决策树而导入的工具包。

接下来，通过信息熵计算划分后的数据集混乱程度，具体如例 3.2 所示。

【例 3.2】 通过信息熵计算划分后的数据集混乱程度。

```
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
myDat,labels=createDataSet()
print(myDat)
print(calcShannonEnt(myDat))
```

输出结果如下所示。

```
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]0.9709505944546686
```

上述结果中，熵的值约等于 0.97。随着数据集复杂程度的增加，熵的值也会增大。读者可以尝试为数据集添加更多的分类，以观察熵的变化。

3.1.4 信息增益与划分数据集

信息增益是指以某特征划分数据集前后的熵的差值。在 3.1.3 节中，本书介绍了度量数据集无序程度的方法，在分类算法中，除了需要计算信息熵，还需要对数据集进行划分，计算划分数据集的熵，从而判断该分类方法是否正确地对数据集进行了划分。通过对按照每种特征划分数据集的结果逐一计算信息熵来找出最优的划分方式。掌握了计算信息增益的方法，就可以计算每一个特征值划分数据集获得的信息增益，获得信息增益最高的特征就是最优特征。

在之前介绍熵的时候已经提到，可以通过熵来表示数据集的不确定性，熵的值越大，数

据集的不确定性就越大。因此，可以根据划分前后数据集熵值的变化来衡量使用当前特征对样本集合 D 划分的效果。划分前数据集 D 的熵是确定的，根据某个特征 A 划分数据集 D，计算划分后的数据子集的熵。信息增益的计算公式如下所示。

$$\mathrm{Gain}(D,A)=\mathrm{Ent}(D)-\mathrm{Ent}(D\mid A)$$

上述表达式中 $\mathrm{Ent}(D)$表示划分前数据集 D 的熵，$\mathrm{Ent}(D\mid A)$表示划分后的数据子集的熵。假定离散特征 A 有 m 个可能的取值$\{a^1,a^2,\cdots,a^m\}$，若使用特征 A 来对样本数据集 D 进行划分，则会产生 m 个分支结点，其中第 m 个分支结点包含了数据集 D 中的所有在特征 A 上取值为 a^m 的样本，记作 D^m。再根据不同的分支结点所包含的样本数量差异，给各分支结点赋予权重$|D^m|/|D|$(样本数越多分支结点的影响越大)。通过这种方式计算出用特征 A 对样本数据集 D 进行划分所获得的“信息增益”，表达式如下所示。

$$\mathrm{Gain}(D,A)=\mathrm{Ent}(D)-\sum_{m=1}^{m}\frac{|D^m|}{|D|}\mathrm{Ent}(D^m)$$

一般情况下，信息增益越大，表示所使用的特征 A 划分得到的分类准确性提升越大。

接下来，通过代码实现数据集划分的方法，具体方法如例 3.3 所示。

【例 3.3】 数据集的划分。

```
def splitDataSet(dataSet, axis, value):
    retDataSet = []                          #创建列表对象引用数据集,防止由于多次调用而
                                             #改变元数据集
    for featVec in dataSet:                  #遍历数据集中的每个元素
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec) #将符合特征的数据抽取出来
    return retDataSet
myDat,labels = createDataSet()
print(myDat)
print(splitDataSet(myDat,0,1))
print(splitDataSet(myDat,0,0))
```

输出结果如下所示。

```
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
[[1, 'yes'], [1, 'yes'], [0, 'no']]
[[1, 'no'], [1, 'no']]
```

上述代码中引入了三个输入参数，分别是待划分数据集 dataSet、划分数据集的特征 axis 和返回的特征值 value。

可以看到例 3.3 的代码中分别使用了 extend()方法和 append()方法来处理列表数据。这两个方法处理数据所得的结果是完全不同的，具体可以参考 Python 基础的相关内容，此处不再赘述。

接下来，通过遍历整个数据集，循环计算信息熵和 splitDataSet()函数来选出最优的特征划分方式。具体方法如例 3.4 所示。

【例 3.4】 选出最优的特征划分方式。

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1      #最后一个元素是当前实例的类别标签
    baseEntropy = calcShannonEnt(dataSet) #计算原始信息熵
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):           #遍历数据集中所有特征
      featList = [example[i] for example in dataSet]  #创建一个列表来存放特征
      uniqueVals = set(featList)           #创建唯一的分类标签列表
      newEntropy = 0.0
      for value in uniqueVals:             #遍历当前特征中所有唯一的特征值
        subDataSet = splitDataSet(dataSet, i, value)
        prob = len(subDataSet)/float(len(dataSet))
        newEntropy += prob * calcShannonEnt(subDataSet) #计算每种划分方式的信息熵
      infoGain = baseEntropy - newEntropy #计算信息增益
      if (infoGain > bestInfoGain):        #将结果与目前所得到的最优划分进行比较
        bestInfoGain = infoGain            #如果结果优于当前最优化分特征,则更新划分特征
        bestFeature = i
    return bestFeature                     #返回最优划分的索引值
myDat,labels = createDataSet()
print(chooseBestFeatureToSplit(myDat))
print(myDat)
```

输出结果如下所示。

```
0
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
```

上述代码中 chooseBestFeatureToSplit()函数用于选取特征,划分数据集并计算出最优的划分特征。完整代码需要整合例 3.1～例 3.3 中的函数。

需要注意的是,在函数中调用的数据需要满足两个基本要求:①数据必须是一种列表,且所有的列表元素都要具有相同的数据长度;②数据的最后一列或每个实例的最后一个元素是当前实例的类别标签。当数据集满足以上两个要求时,即可在函数的第一行判定当前数据集包含多少特征。

上述输出结果表明,列表中第 0 个特征是最优划分特征。此时输出结果为 0 表示根据表 3.1 中“是否可以始终保持在水下生存”这一特征来进行划分是最优的。也就是说,把“可以始终保持在水下生存的海洋生物”划分为一组,“不可以始终保持在水下生存的海洋生物”划分为另一组,是最佳划分方式。根据这一特征进行划分的结果与表 3.1 中的“是否属于鱼类”划分结果最接近,这说明列表中第 0 个特征确实是最优划分特征。

3.2 构建决策树

在构建决策树时,首先要解决的问题是判断出当前数据集中哪个特征在划分数据分类时起决定性作用。为找到决定性的特征,划分出最好的结果,需要对每个特征进行评估。假设已经根据一定的方法选取了待划分的特征,则原始数据集将根据这个特征被划分为几个数据子集。由于特征值可能多于 2 个,因此可能出现多于 2 个分支的划分情况:如果某个

分支下的数据属于同一类型，则无须进一步对数据集进行划分；如果数据子集内的数据不属于同一类型，则需要递归地重复划分数据子集的过程，直到每个数据子集内的数据类型相同。

通过递归方法构建决策树的结束条件是：程序遍历完所有划分数据集的特征，或者每个分支下的所有实例都属于相同的分类。当分支中的所有样本都属于同一分类时，则得到一个叶子结点（任何到达叶子结点的数据必定属于该叶子结点的分类）。在算法开始运行前，可以预先设置最大分组数。在终止递归时，查看算法是否使用了所有特征，若数据集已处理所有特征，但类标签不是唯一的，此时就需要考虑如何优化定义叶子结点。一般情况下，采用多数表决的方法决定该叶子结点的分类。在使用递归算法构建决策树时可能会遇到特征数量并不总是随着划分的进行而减少的问题。

通过在 3.1 节完成的代码中添加如下代码，来返回出现次数最多的分类名称（用于多数表决法决定叶子结点的分类），具体方法如例 3.5 所示。

【例 3.5】 返回出现次数最多的分类名称。

```
#返回出现次数最多的分类名称
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]
```

majorityCnt()函数中使用分类名称的列表，接着创建键值为 classList 中唯一值的数据字典 classCount，该字典对象存储了 classList 中每个类标签出现的频率，最后通过 operator 操作键值排序字典，并返回出现次数最多的分类名称。创建树函数的方法具体如例 3.6 所示。

【例 3.6】 创建树函数。

```
def createTree(dataSet,labels):
    classList = [example[-1] for example in dataSet]   #获取数据集的所有类别
    if classList.count(classList[0]) == len(classList):
      return classList[0]                              #如果数据集的所有类别都相同则
  #不需要划分,使用完所有特征后仍然不能将数据划分到某个类别上,则返回出现次数最多的类别
    if len(dataSet[0]) == 1:
      return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)       #获取数据集中按哪一列进行划分
    bestFeatLabel = labels[bestFeat]                   #bestFeatLabel = 列描述
    myTree = {bestFeatLabel:{}}                        #创建一个字典
    del(labels[bestFeat])                              #删除已计算过的列
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)                       #获取某列所有不重复值
    for value in uniqueVals:
      subLabels = labels[:]
```

```
    myTree[bestFeatLabel][value] = createTree(splitDataSet(
        dataSet, bestFeat, value),subLabels)     #递归
  return myTree
myDat,labels = createDataSet()
myTree = createTree(myDat,labels)
print(myTree)
```

输出结果如下所示。

```
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

上述代码中，createTree()函数中包含了两个输入参数：数据集和标签列表。标签列表包含了数据集中所有特征的标签，算法本身并不需要这个变量，但为了给出明确的数据含义，需要将其作为输入参数导入。代码中首先创建了名为 classList 的列表变量，其中包含数据集中的所有类标签。递归函数的第一个停止条件是若所有类标签完全相同，则直接返回该类标签 classList[0]；第二个停止条件是所有特征被用完时，如果仍然不能将数据集划分为仅包含唯一类别的分组，则采用多数表决的方法决定该叶子结点的分类。

接下来，进行创建树操作。通过字典类型变量 myTree 存储树的所有信息。当前数据集选取的最优特征存储在变量 bestFeat 中，得到列表包含的所有特征值。

最后，遍历当前选择特征包含的所有属性值，在每个数据集划分上递归调用 createTree()函数，得到的返回值将被插入字典变量 myTree 中。在函数终止执行时，字典变量 myTree 中将嵌套很多代表叶子结点信息的字典数据。subLabels = labels[:]复制类标签并将其存储在新列表变量 subLabels 中，该做法可以使得每次调用函数 createTree()时不改变原始列表的内容。

在上述代码的运行结果中，变量 myTree 包含很多树结构信息的嵌套字典，从左开始第一个关键字 flippers 是第一个划分数据集的特征名称，该关键字的值也是另一个数据字典。第二个关键字 no sufacing 特征划分的数据集，这些关键字的值是 flippers 结点的子结点，这些值可能是类标签也可能是另一个数据字典。如果值是类标签，则该子结点是叶子结点；如果值是另一个数据字典，则子结点是一个判断结点，这种格式结构不断重复就构成了整棵树。本例中，这棵树包含了 3 个叶子结点以及 2 个判断结点。

3.3 可视化决策树

决策树的主要优点是直观、易于理解，若不能将其直观地显示出来，就无法体现其优势。本节将讲解如何使用 Matplotlib 库绘制树形图，从而更直观地解释数据信息的含义。

3.3.1 注释结点

Matplotlib 工具库为编程者提供了注解工具——annotations，编程者可以通过它在数据图形中添加文本注释，该注释通常用于解释数据的内容。该工具支持带箭头的画线工具，可在其他恰当的地方指向数据位置并在该处添加描述信息，避免直接在文本上描述。

Matplotlib 的注解功能可以对文字着色并提供了多种图形形状，也可反转箭头将其指向文本框。可以通过以下代码实现相关操作，具体如例 3.7 所示。

【例 3.7】 注解树结点。

```
import matplotlib.pyplot as plt
#定义决策树决策结果的特征，以字典的形式定义
#下面的字典定义也可写作 decisionNode={boxstyle:'sawtooth',fc:'0.8'}
#boxstyle 为文本框的类型，sawtooth 是锯齿形，fc 是边框线粗细
decisionNode = dict(boxstyle="sawtooth", fc="0.8")
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    #annotate 是关于一个数据点的文本
    #nodeTxt 为要显示的文本，centerPt 为文本的中心点，parentPt 为指向文本的点
    createPlot.ax1.annotate(nodeTxt, xy=parentPt, xycoords='axes fraction',
            xytext=centerPt, textcoords='axes fraction',
            va="center", ha="center", bbox=nodeType, arrowprops=arrow_args )
def createPlot():
    fig = plt.figure(1,facecolor='white') # 定义一个画布，背景为白色
    fig.clf() # 把画布清空
    #createPlot.ax1 为全局变量，绘制图像的句柄，subplot 为定义了一个绘图
    #111 表示 figure 中的图有 1 行 1 列，即 1 个，最后的 1 代表第一个图
    #frameon 表示是否绘制坐标轴矩形
    createPlot.ax1 = plt.subplot(111,frameon=False)
    plotNode('a decision node',(0.2,0.2),(0.6,0.8),decisionNode)
    plotNode('a leaf node',(0.6,0.1),(0.8,0.8),leafNode)
    plt.show()
if __name__ == '__main__':
    createPlot()
```

运行程序，结果如图 3.2 所示。

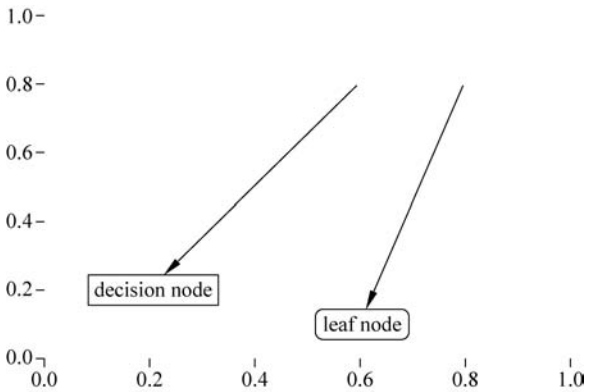


图 3.2 通过 plotNode()函数进行注释

上述代码中首先定义了文本框和箭头格式，然后定义 plotNode()函数执行实际的绘图功能，绘图区域由全局变量 createPlot.ax1 定义。最后定义 createPlot()函数，该函数中首

先创建一个新图形并清空绘图区，然后在绘图区中绘制两个代表不同类型的树结点。

3.3.2 构建完整的注解树

绘制一颗完整的决策树不仅需要有坐标，还要考虑如何放置所有的树结点。需要先确定决策树的结点的个数，以便设置合适的 x 轴长度；再确定决策树的层数，以便设置合适的 y 轴长度。

可通过定义两个新的函数在 3.3.1 节代码的基础上获取叶子结点的数目以及树的层数，这两个函数可命名为 getNumLeafs()与 getTreeDepth()，具体代码如例 3.8 所示。

【例 3.8】 获取叶子结点的数目以及树的层数。

```
def getNumLeafs(myTree):
  numLeafs = 0
  firstStr = list(myTree.keys())[0]          #字典的第一个键,即树的一个结点
  secondDict = myTree[firstStr]              #这个键的值,对应该结点的所有分支
  for key in secondDict.keys():
      if type(secondDict[key]).__name__ == 'dict':
          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':
          thisDepth = 1 + getTreeDepth(secondDict[key])
      else: thisDepth = 1
      if thisDepth > maxDepth: maxDepth = thisDepth
  return maxDepth
```

上述代码定义了 getNumLeafs()函数与 getTreeDepth()函数，不难看出，这两个函数有着相同的结构。函数中演示了如何使用字典类型存储树信息，第一个关键字 firstStr 是第一次划分数据集的类别标签，附带的数值表示子结点的取值，从第一个关键字出发可遍历整棵树的所有子结点。在函数 getNumLeafs()中，使用 type()方法判断子结点为字典类型(即该结点也是一个判断结点)后，则递归调用 getNumLeafs()函数，该过程遍历整棵树，并返回累计叶子结点的个数。函数 getTreeDepth()中将判断结点的个数，当到达叶子结点时从递归调用中返回，并将树深度的变量增加 1。

为避免每次测试代码时都要从数据中重新创建树，可定义一个函数预先存储树信息，具体代码如例 3.9 所示。

【例 3.9】 构建预先存储树信息的函数。

```
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]
if __name__ == '__main__':
    tree = retrieveTree(1)
    leafs = getNumLeafs(tree)
    depth = getTreeDepth(tree)
    print(leafs)
    print(depth)
```

输出结果如下所示。

```
4
3
```

上述结果表明该决策树的叶子结点数为 4，层数为 3。

在当前的 treePlotter.py 文件中，如果直接执行 createPlot()函数，则输出结果仍为图 3.2 中的图形。现对其进行改造以便于绘制树形图，具体修改后的代码如例 3.10 所示。

【例 3.10】 绘制树形图。

```
def plotTree(myTree, parentPt, nodeTxt):
    numLeafs = getNumLeafs(myTree)                    #当前树的叶子数
    depth = getTreeDepth(myTree)                      #没有用到这个变量
    firstSides = list(myTree.keys())
    firstStr = firstSides[0]
    #cntrPt 是文本中心点,parentPt 指向文本中心点
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)            #画分支上的键
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD#从上往下画
    for key in secondDict.keys():
    #如果是字典则是一个判断(内部)结点
        if type(secondDict[key]).__name__ == 'dict':
            plotTree(secondDict[key],cntrPt,str(key))
        else:   #打印叶子结点
            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)
def createPlot(inTree):
    fig = plt.figure(1, facecolor = 'white')
```

```
    fig.clf()
    axprops = dict(xticks=[], yticks=[])          #定义横纵坐标轴
    createPlot.ax1 = plt.subplot(111, frameon=False)
    plotTree.totalW = float(getNumLeafs(inTree))    #全局变量宽度 = 叶子数
    plotTree.totalD = float(getTreeDepth(inTree))   #全局变量高度 = 深度
    #图形的大小是 0-1,0-1
    plotTree.xOff = -0.5/plotTree.totalW;
    #例如绘制 3 个叶子结点,坐标应为 1/3,2/3,3/3
    #但这样会使整个图形偏右,因此初始的 x 值将向左移一点
    plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()
if __name__ == '__main__':
    myTree = retrieveTree(0)
    createPlot(myTree)
```

上述代码在绘制注解树时，不会因为树的结点的增减和深度的增减而导致绘制出来的图形太密集而出现问题。这是因为代码将整棵树的叶子结点数作为份数对整个 x 轴的长度进行了平均切分，将树的深度作为份数对 y 轴长度进行了平均切分。并且，代码将 plotTree.xOff 作为最近绘制的一个叶子结点的 x 坐标，当再一次绘制叶子结点坐标时 plotTree.xOff 才会发生改变；将 plotTree.yOff 作为当前绘制的深度，plotTree.yOff 每递归一层就会减一份（之前利用树的深度将 y 轴平均切分），其他时候利用这两个坐标点计算非叶子结点，通过这两个参数就可以确定一个点坐标，坐标确定后便可以绘制结点。

createPlot()函数调用了 plotTree()与 plotMidText()函数，其中 plotTree()函数在绘制树形图的过程中起关键作用，plotTree()函数中首先计算树的宽和高，全局变量 plotTree.totalW 存储树的宽度，全局变量 plotTree.totalD 存储树的深度。

输出结果如图 3.3 所示。

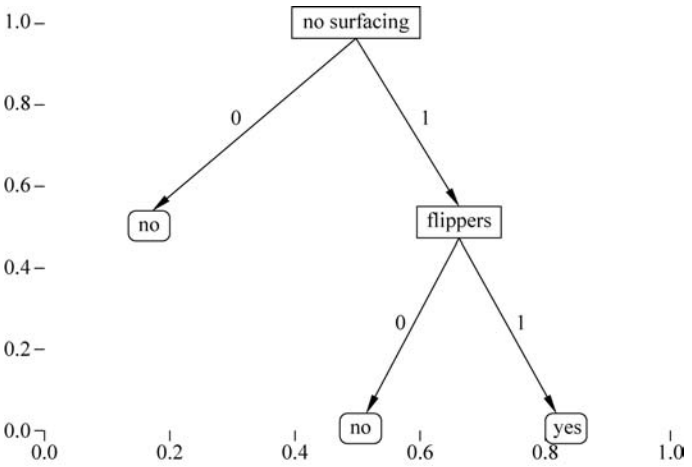


图 3.3　树 0 的树形图绘制

3.4 基尼指数与 CART 算法

除了前面介绍的通过信息熵来衡量集合的无序程度外，另一种常见的度量指标是基尼指数(Gini index)，也称作基尼不纯度(Gini impurity)。基尼指数通过从数据集中随机选取子项来度量该子项被错误分类到其他分组中的概率。

分类与回归树(Classification And Regression Tree，CART)算法通过“基尼指数”来选择划分特征的依据。通过基尼指数来度量数据集合 D 的纯度，度量纯度的表达式如下所示。

$$\mathrm{Gini}(D)=\sum_{i=1}^{n}\sum_{i'\neq i}p(x_i)p(x_{i'})=1-\sum_{i=1}^{n}p(x_i)^2$$

$\mathrm{Gini}(D)$表示从数据集 D 中随机抽取两个样本，其类别标记不一致的概率。因此，$\mathrm{Gini}(D)$的值越小，则数据集 D 的纯度越高。

假定离散特征 A 有 m 个可能的取值$\{a^1,a^2,\cdots,a^m\}$，若使用特定特征 A 来对样本数据集 D 进行划分，则会产生 m 个分支结点，其中第 i 个分支结点包含了数据集 D 中的所有在特征 A 上取值为 a^i 的样本吗，记作 D^i。根据公式计算出 D^i 的信息熵。再根据不同的分支结点所包含的样本数量差异，给各分支结点赋予权重$|D^i|/|D|$(样本数越多，分支结点的影响越大)。通过这种方式计算出用特征 A 对样本数据集 D 进行划分所获得的基尼指数，表达式如下所示。

$$\mathrm{Gini}_{\mathrm{index}}(D,A)=\sum_{i=1}^{m}\frac{|D^i|}{|D|}\mathrm{Gini}(D^i)$$

根据上述表达式，在候选特征集合 A 中，选取使得划分后基尼指数最小的特征作为最优划分特征。

通过 CART 算法构建决策树，具体方法如例 3.11 所示。

【例 3.11】 通过 CART 算法构建决策树。

```
from math import log
import operator
import treePlotter

def calcShannonEnt(dataSet):
    numEntries = len(dataSet) #计算数据集中的实例总数
    labelCounts = {}
  #统计类别出现的次数
  #放到一个数组中 key 表示标签,val 表示个数
    for featVec in dataSet:
      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

def splitDataSet(dataSet, axis, value):
    """
    输入:数据集,选择维度,选择值
    输出:划分数据集
    描述:按照给定特征划分数据集;去除选择维度中等于选择值的项
    """
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reduceFeatVec = featVec[:axis]
            reduceFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reduceFeatVec)
    return retDataSet

def chooseBestFeatureToSplit(dataSet):
    """
    输入:数据集
    输出:最好的划分维度
    描述:选择最好的数据集划分维度
    """
    numFeatures = len(dataSet[0]) - 1                    #特征个数
    bestGini = 999999.0
    bestFeature = -1
    for i in range(numFeatures):
        featList = [example[i] for example in dataSet] #统计第 i 个特征有几种情况
        uniqueVals = set(featList)
        gini = 0.0
        for value in uniqueVals:                         #遍历特征列表
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))  #子集样本个数/总样本个数
            subProb = len(splitDataSet(subDataSet, -1, 'N')) / float(len(subDataSet))
            gini += prob * (1.0 - pow(subProb, 2) - pow(1 - subProb, 2))
        if (gini < bestGini):
            bestGini = gini
            bestFeature = i
    return bestFeature

def majorityCnt(classList):
    """
    输入:分类类别列表
    输出:子结点的分类
    描述:数据集已经处理了所有属性,但是类标签依然不是唯一的,
        采用多数判决的方法决定该子结点的分类
    """
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
```

```
69        classCount[vote] += 1
70    sortedClassCount = sorted(classCount.iteritems(), key = operator.itemgetter(1), reversed =
  True)
71    return sortedClassCount[0][0]
72
73  def createTree(dataSet, labels):
74    """
75    输入:数据集,特征标签
76    输出:决策树
77    描述:递归构建决策树,利用上述的函数
78    """
79   #递归停止条件
80    classList = [example[-1] for example in dataSet] #取所有的类别
81    if classList.count(classList[0]) == len(classList):
82      #只有一个类别时,停止划分
83      return classList[0]
84    if len(dataSet[0]) == 1:
85      #遍历完所有特征时返回出现次数最多的
86      return majorityCnt(classList)
87    bestFeat = chooseBestFeatureToSplit(dataSet) #返回最好的特征
88    bestFeatLabel = labels[bestFeat] #提取特征名称
89    myTree = {bestFeatLabel:{}}   #特征对应的字典
90    del(labels[bestFeat])
91    #得到列表包括结点所有的属性值
92    featValues = [example[bestFeat] for example in dataSet] #取最优特征的种类
93    uniqueVals = set(featValues)
94    for value in uniqueVals:
95      subLabels = labels[:]
96     #递归调用
97      myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),
  subLabels)
98    return myTree
99
100    def classify(inputTree, featLabels, testVec):
101      """
102      输入:决策树,分类标签,测试数据
103      输出:决策结果
104      """
105      firstStr = list(inputTree.keys())[0]
106      secondDict = inputTree[firstStr]
107      featIndex = featLabels.index(firstStr)
108      classLabel = 'N'
109      for key in secondDict.keys():
110          if testVec[featIndex] == key:
111              if type(secondDict[key]).__name__ == 'dict':
112                  classLabel = classify(secondDict[key], featLabels, testVec)
113              else:
114                  classLabel = secondDict[key]
115      return classLabel
116
```

```
117    def classifyAll(inputTree, featLabels, testDataSet):
118        """
119        输入:决策树,分类标签,测试数据集
120        输出:决策结果
121        描述:跑决策树
122        """
123        classLabelAll = []
124        for testVec in testDataSet:
125            classLabelAll.append(classify(inputTree, featLabels, testVec))
126        return classLabelAll
127
128    def storeTree(inputTree, filename):
129        """
130        输入:决策树,保存文件路径
131        输出:决策树文件
132        描述:保存决策树到文件
133        """
134        fw = open(filename, 'wb')
135        pickle.dump(inputTree, fw)
136        fw.close()
137
138    def grabTree(filename):
139        """
140        输入:文件路径名
141        输出:决策树
142        描述:从文件读取决策树
143        """
144        fr = open(filename, 'rb')
145        return pickle.load(fr)
146
147    def createDataSet():
148        """
149        outlook -> 0: sunny | 1: overcast | 2: rain
150        temperature -> 0: hot | 1: mild | 2: cool
151        humidity -> 0: high | 1: normal
152        windy -> 0: false | 1: true
153        """
154        dataSet = [[0, 0, 0, 0, 'N'],
155                   [0, 0, 0, 1, 'N'],
156                   [1, 0, 0, 0, 'Y'],
157                   [2, 1, 0, 0, 'Y'],
158                   [2, 2, 1, 0, 'Y'],
159                   [2, 2, 1, 1, 'N'],
160                   [1, 2, 1, 1, 'Y']]
161        labels = ['outlook', 'temperature', 'humidity', 'windy']
162        return dataSet, labels
163
164    def createTestSet():
165        """
166        outlook -> 0: sunny | 1: overcast | 2: rain
```

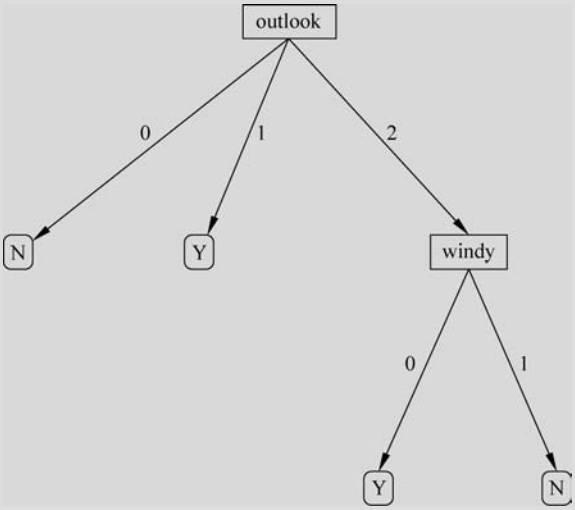
```
        temperature -> 0: hot | 1: mild | 2: cool
        humidity -> 0: high | 1: normal
        windy -> 0: false | 1: true
        """
        testSet = [[0, 1, 0, 0],
                   [0, 2, 1, 0],
                   [2, 1, 1, 0],
                   [0, 1, 1, 1],
                   [1, 1, 0, 1],
                   [1, 0, 1, 0],
                   [2, 1, 0, 1]]
        return testSet

    def main():
        dataSet, labels = createDataSet()
        labels_tmp = labels[:] #复制,createTree 会改变 labels
        desicionTree = createTree(dataSet, labels_tmp)
        #storeTree(desicionTree, 'classifierStorage.txt')
        #desicionTree = grabTree('classifierStorage.txt')
        print('desicionTree:\n', desicionTree)
        treePlotter.createPlot(desicionTree)
        testSet = createTestSet()
        print('classifyResult:\n', classifyAll(desicionTree, labels, testSet))

    if __name__ == '__main__':
        main()
```

输出结果如下所示。

```
desicionTree:
{'outlook': {0: 'N', 1: 'Y', 2: {'windy': {0: 'Y', 1: 'N'}}}}
```




```
classifyResult:
['N', 'N', 'Y', 'N', 'Y', 'Y', 'N']
```

3.5 决策树的剪枝

分类回归树的递归建树的过程中很容易出现数据过拟合的问题。这是因为在构建决策树时，训练数据中存在大量的噪声或孤立点，许多分枝反映的是训练数据中的异常，使用这样的判定树对类别未知的数据进行分类，分类的准确性就会下降。因此，会用到剪枝(pruning)技术。剪枝是指从已经生成的树上裁掉一些子树或叶结点，并将其根结点或父结点作为新的叶子结点，从而简化分类树模型。实现方法是极小化决策树整体的损失函数或代价函数。

决策树常用的剪枝方法有两种：预剪枝(pre-pruning)和后剪枝(post-pruning)。

1. 预剪枝

预剪枝是指在构建决策树的同时进行剪枝，根据一些规则及早地停止树的增长。所谓“规则”可以是指定的树的深度，也可以是指定的结点中样本的个数，还可以是不纯度指标下降到某个指定的幅度。所有决策树的构建方法都是在无法进一步分枝的情况下才会停止创建新的分枝，预剪枝则通过设定某个“规则”提前停止分枝过程，避免过拟合。

预剪枝的核心问题是如何事先指定合适的最大深度。不恰当的最大深度将导致树的生长受限，使决策树的表达式规则趋于一般化，这会限制决策树对新数据进行分类和预测的准确性。除了事先限定决策树的最大深度外，还有另外一个方法可以实现预剪枝操作，那就是采用检验技术对当前结点对应的样本集合进行检验，如果该样本集合的样本数量已小于事先指定的最小允许值，那么停止该结点的生长，并将该结点变为叶子结点。

2. 后剪枝

后剪枝是在决策树生长完成之后进行剪枝，从而得到简化版的决策树。剪枝的过程是对拥有同样父结点的一组结点进行检查，判断如果将其合并，熵的增加量是否小于某一阈值。如果小于指定阈值，则这一组结点可以合并为一个结点。后剪枝是目前更为普遍使用的剪枝方法。目前常见的两种后剪枝技术如下所示。

1) 错误率降低剪枝(Reduced-Error Pruning，REP)

该剪枝方法根据错误率进行剪枝，如果决策树修剪前后子树的错误率没有下降，就可以认为该子树是可以被修剪的。

对于完全决策树中的每一个非叶子结点的子树，尝试着把它替换成一个叶子结点，该叶子结点的类别用子树所覆盖训练样本中存在最多的那个类来代替，这样就产生了一个简化决策树，然后比较这两个决策树在测试数据集中的表现，如果简化决策树在测试数据集中的错误比较少，那么该子树就可以替换成叶子结点。该算法以倒置的方式遍历所有的子树，直至没有任何子树可以替换使得测试数据集的表现得以改进时，算法就可以终止。

2) 悲观剪枝(Pessimistic Error Pruning，PEP)

PEP 剪枝算法是在 C4.5 决策树算法中提出的，把一颗子树(具有多个叶子结点)用一个叶子结点来替代的话，比起 REP 剪枝法，它不需要一个单独的测试数据集。

后剪枝操作是一个边修剪边检验的过程，一般规则标准是：在决策树的不断剪枝操作过程中，将原样本集合或新数据集合作为测试数据，检验决策树对测试数据的预测精度，并计算出相应的错误率，如果剪掉某个子树后的决策树对测试数据的预测精度或其他测度不

降低，那么剪掉该子树。

3.6 本章小结

决策树算法就像带有终止块的流程图，流程图中的终止块对应于决策树中的叶子结点，通过计算数据集中数据的熵寻找最优分类方案。通过 Matplotlib 工具包的注解功能，可以将树结构转变成更加直观的树图模式，方便对决策过程的理解。

决策树算法是机器学习中最简单的一种分类算法，学习决策树算法可以为掌握其他算法做铺垫。

3.7 习　　题

1. 填空题

（1）决策树通常用一棵_____的树结构来表示数据间的逻辑关系，基于_____进行判断，进而得到分类或回归结果。

（2）与 K 近邻算法相比，决策树算法中的_____使得数据形式非常容易理解。

（3）自信息是信息的度量单位，用来衡量单一事件发生时所包含的信息量多寡，它的单位是_____。

（4）信息增益是指以某特征划分数据集前后的_____的差值。

2. 选择题

（1）以下选项不属于决策树算法的特点的是（　　）。

A. 善于处理数值型和标称型数据　　B. 善于解决分类问题

C. 善于处理具有不相关特征的数据　　D. 善于处理数据缺失的数据集

（2）概率越小的事件发生后所包含的自信息（　　），概率越大的事件发生后所包含的自信息（　　）。

A. 越少　越多　　B. 越多　越少

C. 越少　越少　　D. 自信息与事件发生的概率无关

（3）根据本章所介绍的知识，在决策树算法中，可以通过（　　）计算数据集混乱程度。

A. 自信息　　B. 基尼指数

C. 信息熵　　D. B 和 C 都对

3. 思考题

简述计算信息增益在决策树算法中的意义。