机器学习 **Lecture 4**

决策树

一、 决策树原理

1. Introduction

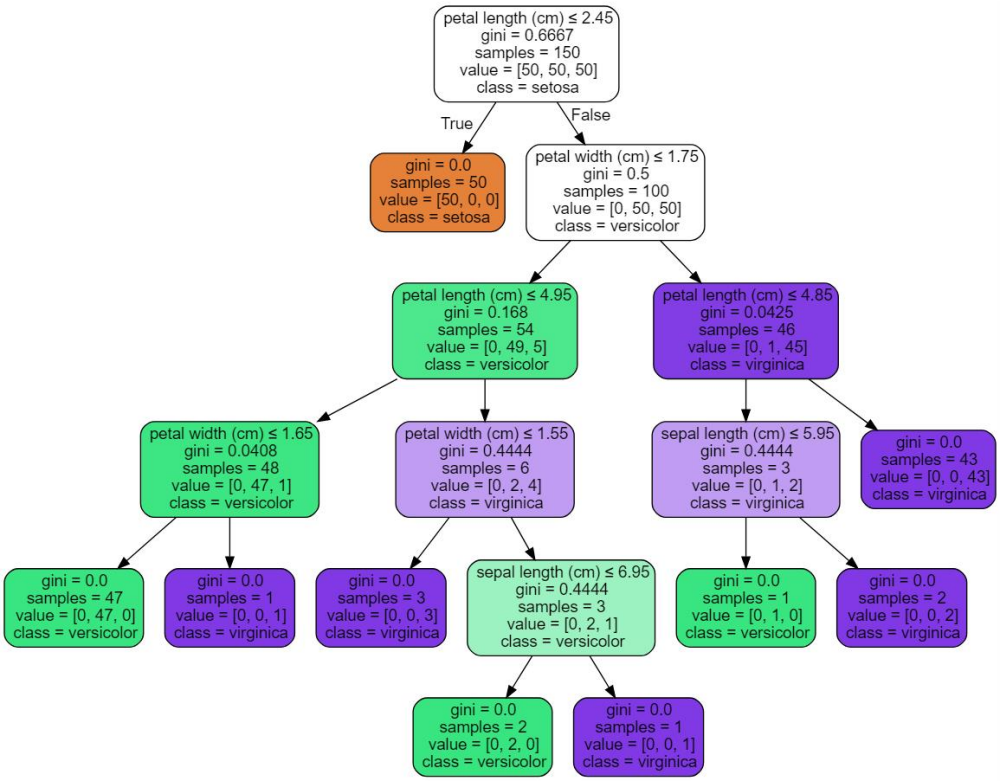


图 1: 决策树

决策树是从训练数据中学习得出一个树状结构的判别模型。决策树通过做出一系列决策来对数据进行划分。

决策树的核心在于如何选择划分字段。一种常见的方法是通过信息增益来进行选择，信息增益越大，则划分后的数据越纯净。

2. 信息熵、条件熵与信息增益

Defination 1 (信息熵). 信息熵用来表示信息量的大小，信息量越大，对应的熵值就越大，分类越不“纯净”。

$$\mathrm{Ent}(D) = -\sum_{k=1}^{|\mathcal{Y}|} p_k \log_2 p_k$$

Defination 2 (条件熵). 条件熵表示已知 a 事件发生情况下 D 的信息熵

$$\mathrm{Ent}(D|a) = \sum_{v=1}^{\mathcal{V}} \frac{|D^v|}{|D|}\mathrm{Ent}(D^v)$$

其中，a 有 v 个属性，D 在 a^v 属性上的数据集为 D^v

Defination 3 (信息增益). 信息增益表明在发生事件 A 前后信息熵的差异。信息增益越大，表明 A 对 D 的影响越大，划分后的数据集越纯净。

$$\mathrm{Gain}(D|a) = \mathrm{Ent}(D) - \mathrm{Ent}(D|a)$$

但是通过信息增益划分有一个缺点，即即信息增益会偏向于取值较多的字段。为了克服信息增益指标的缺点，我们可以引入信息增益率：

Defination 4 (信息增益率).

$$\mathrm{Gain\_Ratiao}_A(D) = \frac{\mathrm{Gain}_A(D)}{\mathrm{IV}(a)}$$

$$\mathrm{IV}(A) = -\sum_{v=1}^{V} \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

其中，$\mathrm{IV}(A)$ 是事件 A 的信息熵

3. 基尼指数

由于信息增益率只能对离散的因变量进行分类，为了克服该缺点，可以引入 Gini 指数来选择划分属性

Defination 5 (Gini 指数).

$$\begin{aligned}\mathrm{Gini}(D) &= \sum_{k=1}^{|\mathcal{Y}|} p_k(1-p_k) \\ &= 1 - \sum_{k=1}^{|\mathcal{Y}|} p_k^2 \\ &= 1 - \sum_{k=1}^{|\mathcal{Y}|} (\frac{|C_k|}{|D|})^2\end{aligned}$$

Defination 6 (条件 Gini 指数). 类似条件熵，条件 *Gini* 指数可以定义为

$$\mathrm{Gini}(D|a) = \sum_{v=1}^{\mathcal{V}} \frac{|D^v|}{|D|}\mathrm{Gini}(D^v)$$

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

图 2: 西瓜数据集

Example 1. 对于西瓜数据集（图2），计算色泽的信息增益与基尼指数增益。信息增益

$$
\begin{aligned}
\mathrm{Ent}(D) &= -\frac{8}{17}\log_2(\frac{8}{17}) - \frac{9}{17}\log_2(\frac{9}{17}) = 0.9975 \\
\mathrm{Ent}(D|\text{色泽}) &= (\frac{6}{17})(-\frac{3}{6}\log\frac{3}{6} - \frac{3}{6}\log\frac{3}{6}) \\
&\quad + (\frac{6}{17})(-\frac{4}{6}\log\frac{4}{6} - \frac{2}{6}\log\frac{2}{6}) \\
&\quad + \frac{5}{17}(-\frac{1}{5}\log\frac{1}{5} - \frac{4}{5}\log\frac{4}{5}) \\
&= 0.353 + 0.324 + 0.212 = 0.889 \\
\mathrm{Gain}(D, \text{色泽}) &= \mathrm{Ent}(D) - \mathrm{Ent}(D|\text{色泽}) = 0.109
\end{aligned}
$$

基尼指数增益

$$\begin{aligned}\text{Gini}(D) &= 1-(\frac{8}{17}^2+\frac{9}{17}^2)=0.498\\ \text{Gini}(D|\text{色泽}) &= \frac{6}{17}(1-(\frac{3}{6}^2+\frac{3}{6}^2))\\ &\quad+\frac{6}{17}(1-(\frac{4}{6}^2+\frac{2}{6}^2))\\ &\quad+\frac{5}{17}(1-(\frac{1}{5}^2+\frac{4}{5}^2))\\ &= 0.427\\ \text{Gini}(D,\text{色泽}) &= 0.071\end{aligned}$$

4. 剪枝

剪枝是决策树应对过拟合的主要手段。剪枝分为预剪枝和后剪枝。

Defination 7 (预剪枝)**.** 将数据集划分为训练集和验证集，每次选择字段 a 并生成 V 个叶节点后，在验证集上计算精度。若精度大于划分前，则继续划分，否则停止划分

优点

- 降低过拟合风险
- 显著减少训练时间和测试时间开销

缺点

- 无法达到全局最优解

Defination 8 (后剪枝)**.** 将数据集划分为训练集和验证集，从训练集学习到一颗完整的决策树。按照分类节点从外往里的顺序，每次将分类节点替换为叶节点，类别为该节点下最多的类。在验证集中计算精度，如果精度提升，则剪枝，否则不剪枝。

优点

- 欠拟合风险小
- 泛化性能优于预剪枝的决策树

缺点

- 训练时间开销和测试时间开销大于预剪枝决策树

二、 算法

1. 生成决策树

生成决策树的算法如 Algorithm1所示

Algorithm 1 决策树

Input: $D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}$； $A = \{a_1, a_2, \cdots, a_d\}$

Def TreeGenerate(D, A)
生成节点 node
if D 中样本属于同一类别 C **then** ▷ 已经分类正确
    将 node 标记为 C 类叶节点；**return**
end if
if $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then** ▷ 不可再分条件
    将 node 标记为叶节点，其类别标记为 D 中样本数最多的类；**return**
end if
从 A 中选择划分最优属性 a_*
for a_* 的每一个值 a_*^v **do**
    为 node 生成一个分支；令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集
    if $D_v = \varnothing$ **then**
        将分支节点标记为叶节点，其类别标记为 D 中样本中最多的类；**return** ▷ 为了泛化性
    else
        以 TreeGenerate($D_v, A \setminus \{a_*\}$) 为分支节点
    end if
end for

Output: 以 node 为结点的一颗决策树

表 1: 节点类型

类别	属性				
	class	class_dict	attr	attrName	children
根节点		✓		✓	✓
分类节点		✓	✓	✓	✓
叶节点	✓		✓		

(1) 程序设计

决策树

共有 3 类结点：根节点，分类节点和叶节点。其属性值有一些差异，如表1所示。所有的节点共用一个 Node 类，但是会在 Node 类中表明该节点属于哪种类型。

其中，class 用于储存叶节点的分类类型，如是好瓜或者坏瓜，并用于决策树的预测。class_dict 用于储存在该节点对应的训练集中，各个分类的数量，如有 3 个好瓜 1 个坏瓜，在进行剪枝操作时，可以较为方便地判断剪枝后该节点的分类类型。attrName 储存属性名称，如“色泽”。attr 储存某一属性的值，如“青绿”，表示从父节点划分到该节点依据的是“色泽”=“青绿”。children 储存该节点的子节点。

函数

为了方便程序编写，共用到了 7 个函数，具体功能如表2所示

表 2: 函数

函数名称	注释	
	对应伪代码行数	功能
isSame	3	判断 D 中的数据是否是同一个类别
sameValue	6	判断 D 中是否所有的样本都相同
countClass	7	统计 D 中每个类别的个数
countAttr	10	统计 D 中指定分类属性的各个属性的个数
IF	9	计算信息熵
CE	9	计算每个属性的信息增益
IFD	9	计算 D 上的信息熵

2. 后剪枝

Algorithm 2 后剪枝

Input: 根节点 root

遍历决策树，获取所有分类节点和对应深度 nodeList
while nodeList 不为空 **do**
    选出 nodeList 中深度最深的节点 node
    计算剪枝前精度 acc1
    复制 node 中的内容 ▷ 用于恢复节点内容
    剪枝: 将 class 赋值为 class_dict 中数量最大的类，将类型修改为叶节点，清空子节点
    计算剪枝后精度 acc2
    if acc2 $\leq$ acc1 **then**
        恢复节点内容
    end if
    从 nodeList 中删除 node 节点
end while

Output: root 节点

后剪枝的算法如 Algorithm2所示

3. 预剪枝

预剪枝的代码如 Algorism3所示，预剪枝是在决策树的基础上进行修改的，在递归产生分支之前将程序截断，只往下分支一层，并比较精度。如果精度提升，则返回 True, 继续递归产生分支，否则返回 False，程序结束。Algorism3的代码在 Algorism1的 9-10 行之间。

三、 运行结果

在西瓜数据集上运行了决策树，其结果如图3所示

Algorithm 3 预剪枝

计算分支前的精度 acc1
for a_* 的每一个值 a_*^v **do**
    为 node 生成一个分支；令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集
    if $D_v = \varnothing$ **then**
        将分支节点标记为叶节点，其类别标记为 D 中样本中最多的类；**return** ▷ 为了泛化性
    else
        生成一个叶节点 node，属性赋值为 a_*^v，类别标记为样本中最多的类别
    end if
end for
计算分之后精度 acc2
if acc2>acc1 **then**
    return True
else
    return False
end if

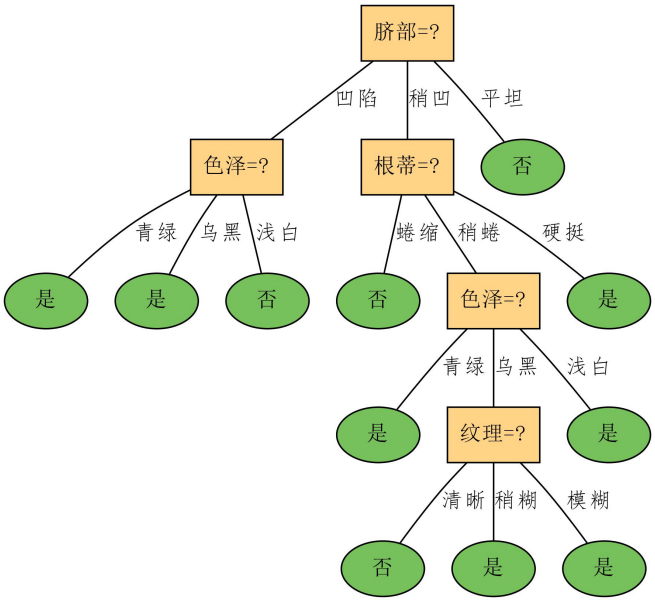
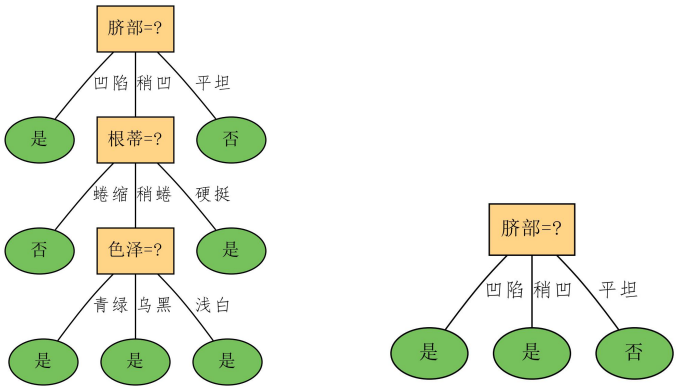


图 3: 不剪枝

附录

A 代码

```
import pandas as pd
```



(a) (b)

图 4: 预剪枝与后剪枝. (a) 后剪枝. (b) 预剪枝.

```python
import numpy as np
import graphviz as gz


class Node:
    def __init__(self, class_=None, type=None, attr=None, attr_name=None):
        self.children = []      # 子节点
        self.class_ = class_    # 叶节点类别
        self.type = type        # 节点类型
        self.attr = attr        # 父节点到该节点的属性
        self.class_dict = {}    # 节点类别字典
        self.attrName = attr_name
    def copy(self, node):
        node.children = self.children
        node.class_ = self.class_
        node.type = self.type
        node.attr = self.attr
        node.class_dict = self.class_dict
        node.attrName = self.attrName
    def print(self):
        print(self.class_,self.class_dict,self.attrName,self.type,self.children)


class Decision_Tree:
    def __init__(self, data, val, A, plot=True):
        self.data = data
        self.val = val
        self.attr_num = {}
        self.plot = plot
        self.node = Node()
        for i in range(len(A)):
```

```
33          self.attr_num[A[i]]=i
34
35      def train(self, prune = None):
36          if prune is None:
37              self.node = self.TreeGenerate(self.data,list(self.attr_num.keys()),
38                                            attr=None,root=True)
39          elif prune == 'post':
40              self.node = self.TreeGenerate(self.data,list(self.attr_num.keys()),
41                                            attr=None,root=True)
42              self.postPruning()
43          elif prune == 'pre':
44              self.node = self.TreeGenerate(self.data,list(self.attr_num.keys()),
45                                            attr=None,root=True,pre=True)
46
47          if self.plot:
48              graph = gz.Graph()
49              self.draw_DT(graph, self.node, 0)
50              graph.view()
51
52      def predict_node(self, node, x):
53          attrName = node.attrName
54          if len(node.children)==0:
55              return node.class_
56          for child in node.children:
57              if child.attr == x[self.attr_num[attrName]]:
58                  class_ = self.predict_node(child,x)
59                  return class_
60      def predict(self, x):
61          class_ = self.predict_node(self.node, x)
62          return class_
63
64      def accuracy(self):
65          cnt = 0
66          for i in range(len(self.val)):
67              if self.predict(self.val[i,:-1]) == self.val[i,-1]:
68                  cnt+=1
69          return cnt/len(self.val)
70
71      def postPruning(self):
72          divideList = self.getDivideNode(self.node,0)
73          print(divideList)
74          while len(divideList)>0:
75              depth = 0
76              index = 0
77              for i in range(len(divideList)):
78                  if divideList[i][1]>depth:
79                      depth = divideList[i][1]
80                      index = i
81              acc1 = self.accuracy()
82              node = divideList[index][0]
```

```
83            node_copy = Node()
84            node.copy(node_copy)
85            # ------------------------------------------------- #
86            # 剪枝
87            # ------------------------------------------------- #
88            node.type = 'leaf'
89            class_dict = node.class_dict
90            class_ = max(class_dict, key=class_dict.get)
91            node.class_ = class_
92            node.children = []
93            # ------------------------------------------------- #
94            # 比较剪枝前后的准确率,如果效果没有变好，则恢复
95            # ------------------------------------------------- #
96            acc2 = self.accuracy()
97            if acc2 <= acc1:
98                node_copy.copy(node)
99                print(f'acc:{acc1},不剪枝,节点信息：字段 {node.attr},深度
                      {divideList[index][1]}')
100           else:
101               print(f'acc:{acc2},剪枝,节点信息：字段 {node.attr},深度
                      {divideList[index][1]}')
102           divideList.pop(index)
103       # graph = gz.Graph()
104       # self.draw_DT(graph,self.node,0)
105       # graph.view()
106       return self.node
107
108
109   def getDivideNode(self, node, depth):
110       devideNodeList = []
111       if len(node.children)>0:
112           for child in node.children:
113               devideNodeList.extend(self.getDivideNode(child, depth+1))
114       if node.type == 'divide':
115           devideNodeList.append((node,depth))
116           return devideNodeList
117       return devideNodeList
118
119   def TreeGenerate(self, D, A, attr, root, pre=False):
120       node = Node()
121       if root:
122           self.node=node
123       node.attr = attr
124       # ------------------------------------------------- #
125       # 判断是否都是同一个类，如果都是同一个类，分类完成，return
126       # ------------------------------------------------- #
127       if self.isSame(D):
128           node.class_ = D[0][-1]
129           node.type = 'leaf'
130           return node
```

```
        # ---------------------------------------------------- #
        # 判断是否可以再分，如果不能再分，分类完成，return
        # ---------------------------------------------------- #
        if len(A)==0 or self.sameValue(D):
            node.type = 'leaf'
            count = self.countClass(D)
            class_ = max(count, key=count.get)
            node.class_ = class_
            return node
        # ---------------------------------------------------- #
        # 计算类别的分布 用于后剪枝的时候判断divide节点
        # 变成叶节点的时候应该归属到哪一个类
        # ---------------------------------------------------- #
        node.class_dict = self.countClass(D)
        # ---------------------------------------------------- #
        # 计算样本D上的信息熵
        # ---------------------------------------------------- #
        ifd = self.IFD(D)
        # ---------------------------------------------------- #
        # 挑选信息增益最大的属性a*
        # ---------------------------------------------------- #
        best_attr = ['',0]
        for attr in A:
            gain = ifd - self.CE(D,self.attr_num[attr])
            if gain > best_attr[1]:
                best_attr = [attr,gain]
        # ---------------------------------------------------- #
        # 统计D中a*每个类别的数量
        # 注：a*的类别应该从整个数据集中统计，而非D中，但是数量要在D中统计
        # ---------------------------------------------------- #
        attr_dict = self.countAttr(D,self.attr_num[best_attr[0]])
        print(attr_dict)
        # ---------------------------------------------------- #
        # 设置node的属性（分类节点，按照什么属性分类）
        # ---------------------------------------------------- #
        node.type = 'divide'
        node.attrName = best_attr[0]
        # ---------------------------------------------------- #
        # 对于分类属性的每一个类，判断是否为空
        # 如果为空则新分支标记为D中最多的类，否则重复上述步骤
        # ---------------------------------------------------- #
        ifContinue = True
        if pre:
            ifContinue = self.prePruning(node,attr_dict,best_attr,D)
        if ifContinue:
            for attr in attr_dict.keys():
                if attr_dict[attr] == 0:
                    count = self.countClass(D)
                    class_ = max(count, key=count.get)
                    node.children.append(Node(class_=class_,type='leaf',attr=attr))
```

```
            else:
                Dv = D[np.where(D[:,self.attr_num[best_attr[0]]]==attr),:][0]
                A2 = A.copy()
                A2.remove(best_attr[0])
                print(best_attr[0],attr)
                print(A2)
                print('# ------------------------------------------------- #')
                node.children.append(self.TreeGenerate(Dv,A2,attr,False,pre=pre))
        return node

    def prePruning(self, node, attr_dict, best_attr, D):
        if node.class_ is None:
            node.class_ = max(node.class_dict, key=node.class_dict.get)
        acc1 = self.accuracy()
        for attr in attr_dict.keys():
            if attr_dict[attr] == 0:
                count = self.countClass(D)
                class_ = max(count, key=count.get)
                node.children.append(Node(class_=class_,type='leaf',attr=attr))
            else:
                Dv = D[np.where(D[:,self.attr_num[best_attr[0]]]==attr),:][0]
                class_dict = self.countClass(Dv)
                class_ = max(class_dict, key=class_dict.get)
                child = Node(type='leaf',class_=class_,attr=attr)
                node.children.append(child)
        # graph = gz.Graph()
        # self.draw_DT(graph, self.node, 0)
        # graph.view()
        acc2 = self.accuracy()
        node.children = []
        if acc2>acc1:
            print(f'划分前: {acc1}, 划分后: {acc2}, 继续划分')
            return True
        else:
            print(f'划分前: {acc1}, 划分后: {acc2}, 禁止划分')
            return False

    def isSame(self, D):
        '''
        判断D中的数据是否是同一个类别
        :param D: 数据集的一个子集
        :return: bool
        '''
        class_ = D[0][-1]
        for i in range(len(D)):
            if D[i][-1] != class_:
                return False
        return True

    def countClass(self, D):
```

```
        '''
        统计D中每个类别的个数
        :param D: 数据集的一个子集
        :return: dict{class:cnt}
        '''
        class_dict = {}
        for i in range(len(D)):
            class_ = D[i][-1]
            class_dict.setdefault(class_,0)
            class_dict[class_]+=1
        return class_dict

    def sameValue(self, D):
        '''
        判断D中是否所有的样本都相同
        :param D: 数据集的一个子集
        :return: bool
        '''
        v = D[0]
        for i in range(D.shape[0]):
            if (v != D[i]).any():
                return False
        return True

    def countAttr(self, D, attr_idx):
        '''
        统计D中指定分类属性的各个属性的个数
        :param D: 数据集的一个子集
        :param attr_idx: 属性编号，输入时生成
        :return: dict{attr:cnt}
        '''
        attr_dict = {}
        for i in range(len(self.data)):
            attr = self.data[i][attr_idx]
            attr_dict.setdefault(attr,0)
        for i in range(len(D)):
            attr = D[i][attr_idx]
            # attr_dict.setdefault(attr, 0)
            attr_dict[attr] += 1
        return attr_dict

    def IF(self, x):
        '''
        计算信息熵
        :param x: ndarray
        :return: 信息熵
        '''
        x[np.where(x==0)] = 1
        return -np.sum(x*np.log2(x))

```

```
281     def CE(self, D, attr_idx):
282         '''
283         计算每个属性的信息增益
284         :param D: 数据集的一个子集
285         :param attr_idx: 属性编号，输入时生成
286         :return: 信息增益
287         '''
288         attr_dict = self.countAttr(D,attr_idx)
289         EF = 0
290         for attr in attr_dict.keys():
291             Dv = D[np.where(D[:,attr_idx]==attr),:][0]
292             ifd = self.IFD(Dv)
293             EF += len(Dv)/len(D)*ifd
294         return EF
295     def IFD(self, D):
296         '''
297         计算D上的信息熵
298         :param D: 数据集的一个子集
299         :return: 信息熵
300         '''
301         class_dict = self.countClass(D)
302         class_dict = np.array(list(class_dict.values()))
303         class_dict = class_dict / np.sum(class_dict)
304         return self.IF(class_dict)
305 
306     def draw_DT(self, graph, node, nodeid):
307         '''
308             通过graphviz进行决策树可视化（递归）
309             :param graph: 图
310             :param node: 上一级节点（决策树的节点）
311             :param nodeid: 上一级节点编号（graph的节点编号）
312             :return: nodeid
313             '''
314         if len(node.children)==0:
315             graph.node(str(nodeid), node.class_, fontname='SimSun',
316                        style='filled',color='black',fillcolor='#78C25E')
317         else:
318             graph.node(str(nodeid), node.attrName + '=?',
319                        fontname='SimSun',shape='box',
320                        style='filled',color='black',fillcolor='#FFD588')
321         node.print()
322         curr_nodeid = nodeid
323         # print(node.class_,nodeid)
324         if node.children:
325             for child in node.children:
326                 child_id = nodeid + 1
327                 nodeid = self.draw_DT(graph, child, child_id)
328                 graph.edge(str(curr_nodeid), str(child_id), label=child.attr,
                        fontname='FangSong')
329         return nodeid
```