机器学习实验报告

决策树

朱天泽

（日期：2022 年 4 月 4 日）

摘要 在《机器学习》第四章中，我学习了决策树模型。在此次实验中，我尝试用 DFS 和 BFS 实现了多个用于分类任务的决策树模型，包括针对离散值的单变量决策树、基于对数几率回归的多变量决策树。在训练好决策树模型的基础上，我使用了 4 个 UCI 数据集，比较了未剪枝、预剪枝、后剪枝三种类型决策树的准确率。此外，我尝试使用了机器学习库 scikit-learn，同自己实现的算法进行了比较。
关键词 决策树；剪枝；分类；搜索；scikit-learn

1 习题 1

1.1 编程题目理解

题目要求：任意选择 4 个 UCI 数据集，对基于信息增益划分选择（ID3）、基于基尼指数划分选择（CART），基于对率回归划分选择的决策树算法（包括未剪枝、预剪枝、后剪枝三种）进行实验比较。

ID3、CART 两种划分选择，可用于实现单变量的决策树；对率回归是一个线性分类模型，基于对率回归，可以实现多变量决策树。在实现未剪枝决策树的基础上，还需要对决策树进行前剪枝或后剪枝。

1.2 决策树模型原理阐述

1.2.1 决策树构建过程

一般的，一棵决策树包含一个根结点、若干个内部结点和若干个叶结点；叶结点对应于决策结果，其它每个结点则对应于一个属性测试；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。从根结点到每个叶结点的路径对应了一个判定测试序列。决策树学习的目的是为了产生一棵泛化能力强，即处理未见示例能力强的决策树，其基本流程遵循简单且直观的“分而治之”策略。

就如算法1，决策树的生成是一个递归过程。在决策树基本算法中，有三种情形会导致递归返回：

1. 当前节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，或是所有样本在所有属性上取值相同，无法划分；
3. 当前节点包含的样本集合为空，不能划分。

算法 1 决策树学习基本算法

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_m, y_m)\}$；属性集 $A = \{a_1, a_2, \cdots, a_d\}$。

输出： 决策树的根节点 root

function TreeGenerate(D, A) 生成结点 node;
  if D 中样本全属于同一类别 C **then**
    将 node 标记为 C 类叶结点；**return**
  end if
  if $A \neq \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
    将 node 标记为叶结点，其类别标记为 D 中样本最多的类；**return**
  end if
  从 A 中选择最优划分属性 a_*；
  for a_* 的每个值 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 为空 **then**
      将分支结点标记为叶结点，其类别标记为 D 中样本最多的类；**return**
    else
      以 TreeGenerate(D_v, $A \setminus \{a_*\}$) 为分支结点
    end if
  end for
end function

在第 2 种情形下，我们把当前结点标记为叶结点，并将其类别设定为该结点所含样本最多的类别；在第 3 种情形下，同样把当前结点标记为叶结点，但将其类别设定为其父结点所含样本最多的类别。这两种情形的处理有实质不同：情形 2 是在利用当前结点的后验分布，而情形 3 则是把父结点的样本分布作为当前结点的先验分布。

1.2.2 划分选择

由算法1可以看出，决策树学习的关键是如何选择最优划分属性。一般而言，随着划分过程不断进行，我们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。在本次实验，我实现了基于信息增益和基尼指数的决策树。

信息增益 ID3 决策树学习算法以信息增益为准则来选择划分属性。

信息熵是度量样本集合纯度最常用的一种指标。假定当前样本集合 D 中第 k 类样本所占的比例 p_k $(k = 1, 2, \cdot, |\mathcal{Y}|)$，则 D 的信息熵定义为

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

$\mathrm{Ent}(D)$ 的值越小，则 D 的纯度越高。

假定离散属性 a 有 V 个可能的取值 $\{a^1, a^2, \cdots, a^V\}$，若使用 a 来对样本集 D 进行划分，则会产生 V 个分支结点，其中第 v 个分支结点包含了 D 中所有在 a 取值上为 a^v 的样本，记为 D^v。我们可

根据式(1)计算出 D^v 的信息熵，再考虑到不同的分支结点所包含的样本数不同，给分支结点赋予权重 $\frac{|D^v|}{|D|}$，即样本数越多的分支结点的影响越大，于是可计算出用属性 a 对样本集 D 进行划分所获得的“信息增益”

$$\operatorname{Gain}(D,a)=\operatorname{Ent}(D)-\sum_{v=1}^{V}\frac{|D^v|}{|D|}\operatorname{Ent}\left(D^v\right) \tag{2}$$

一般而言，信息增益越大，则意味着使用属性 a 来进行划分所获得的“纯度提升”越大。因此，我们可用信息增益来进行决策树的划分属性选择，即在算法1中，$a_*=\underset{a\in A}{\arg\max}\operatorname{Gain}(D,a)$。

基尼指数 CART 决策树使用“基尼指数”来选择划分属性。

采用与式(1)相同的符号，数据集 D 的纯度可用基尼值来度量：

$$\begin{aligned}\operatorname{Gini}(D)&=\sum_{k=1}^{|\mathcal{Y}|}\sum_{k'\neq k}p_kp_{k'}\\&=1-\sum_{k=1}^{|\mathcal{Y}|}p_k^2\end{aligned} \tag{3}$$

直观来说，$\operatorname{Gini}(D)$ 反映了从数据集 D 中随机抽取两个样本，其类别标记不一致的概率。因此，$\operatorname{Gini}(D)$ 越小，则数据集 D 的纯度越高。

采用与式(2)相同的符号表示，属性 a 的基尼指数定义为

$$\operatorname{Gini\_index}(D,a)=\sum_{v=1}^{V}\frac{|D^v|}{|D|}\operatorname{Gini}\left(D^v\right) \tag{4}$$

于是，我们在候选属性集合 A 中，选择那个是的划分后基尼指数最小的属性作为最优划分属性，即

$$a_*=\underset{a\in A}{\arg\min}\operatorname{Gini\_index}(D,a) \tag{5}$$

1.2.3 剪枝处理

在决策树学习中，为了尽可能正确分类训练样本，结点划分过程将不断重复，有时会造成决策树分支过多，以致于把训练集自身的一些特点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”。预剪枝是指在决策树生成过程中，对每个结点在划分前先进行估计，若当前结点的划分不能带来决策树泛化性能提升，则停止划分并将当前结点标记为叶结点；后剪枝则是先从训练集生成一棵完整的决策树，然后自底向上地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提升，则将该子树替换为叶结点。

决策树的泛化性能需要使用测试集来评估，在预剪枝和后剪枝的过程中，若添加或删除结点能提高测试集的正确率，则执行操作。

1.2.4 多变量决策树

若我们把每个属性视为坐标空间中的一个坐标轴，则 d 个属性描述的样本就对应了 d 维空间中的一个数据点，对样本分类则意味着在这个坐标空间中寻找不同类样本之间的分类边界。决策树所形成的分类边界有一个明显的特点：轴平行，即它的分类边界由若干个与坐标轴平行的分段组成。因为每一次决策只考查一个属性将样本进行划分。

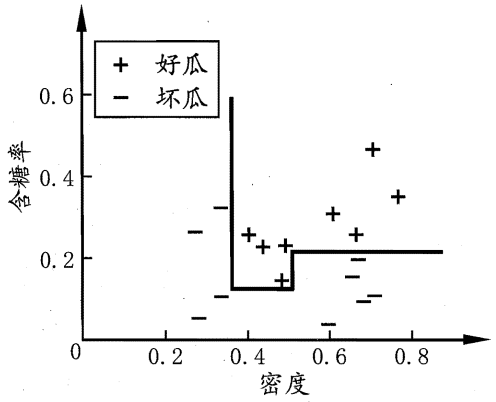

图 1: 决策树对应的分类边界

若能使用斜的划分边界，如图2中红色线段所示，决策树模型将大为简化。“多变量决策树”就是能实现这样的“斜划分”甚至更复杂划分的决策树。

在多分类决策树中，非叶结点不再是仅针对某个属性，而是对属性的线性组合进行测试；换言之，每个非叶结点是一个形如 $\sum_{i=1}^{d} w_i a_i = t$ 的线性分类器，其中 w_i 是属性 a_i 的权重，w_i 和 t 可在该结点包含的样本集和属性集上学得。

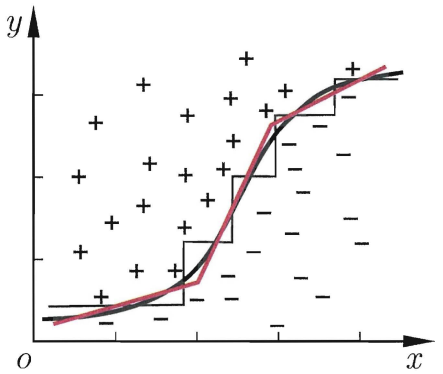

图 2: 决策树复杂分类边界的分段近似

总之，在多变量决策树的学习过程中，不是为每个非叶结点寻找一个最优划分属性，而是试图建立一个合适的线性分类器。在此次实验中，我使用对率回归构建多变量决策树。

1.3 算法设计思路

1.3.1 数据结构设计

这里使用NetworkX创建树结构。

NetworkX 可赋予结点和边任意属性，这里为边和结点赋予属性 label。如图3，树的非叶结点的 label 标记选择划分的属性，叶结点 label 标记类别，边的 label 标记属性值。

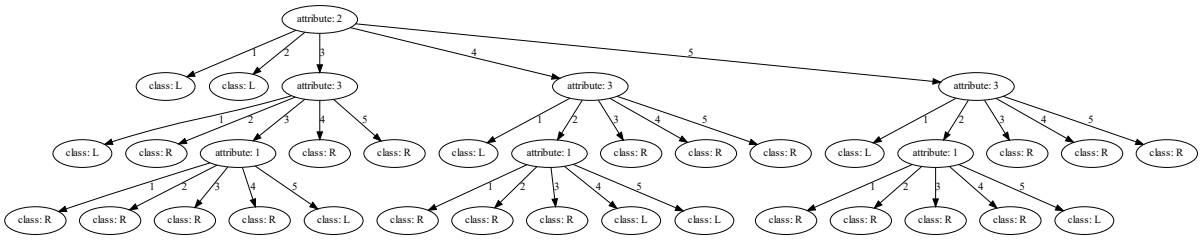

图 3: 数据结构设计

1.3.2 建树过程

基于算法1，我们可以将其加以改造为算法2，实现建树过程。递归的建树过程是 DFS，我们可将其再改造为如算法3的 BFS 过程。

1.3.3 多变量决策树

在本次实验中，结点上用对数几率回归作为线性分类器，对决策树结点上的样本进行二分类。如算法4所述，二分类后的样本会产生两个分支结点，若分支结点上的样本属于同一类别，或 D 中样本不可进行二分类，则结束递归；否则继续扩展分支结点。

1.3.4 预测方法

决策树学习完成后，可对样本进行预测。在预测样本真实类别时，从根结点出发，根据结点标记的划分属性，走对应划分属性值的边，到达子结点。

以图4为例，一个西瓜的纹理清晰、根蒂稍蜷、色泽浅白，它的预测结果是好瓜。其决策路径在图4中用红色标出。

1.3.5 预剪枝

先剪枝在建树过程中进行，使用 BFS 方式构建较为方便。在算法3生成分支结点前，需要暂时生成两棵树 T_s 和 T_d，其中 T_s 为以标记当前结点 cur 为叶结点的树，T_d 为生成 cur 分支结点的树。使用测试集对两棵决策树进行性能评估，若 T_d 的正确率更高，则生成分支结点继续搜索，否则在当前结点停止扩展。

算法 2 决策树学习 DFS

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_m, y_m)\}$；属性集 $A = \{a_1, a_2, \cdots, a_d\}$；当前结点索引 cur。

function TreeGenerate(D, A, cur)
  if D 中样本全属于同一类别 C **then**
    将 cur 标记为 C 类叶结点，其 label 为 C；**return**
  end if
  if $A \neq \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
    将 cur 标记为叶结点，其 label 标记为 D 中样本最多的类；**return**
  end if
  从 A 中选择最优划分属性 a_*；
  将 cur 的 label 标记为 a_*；
  for a_* 的每个值 a_*^v **do**
    为 cur 生成一个分支结点 nxt；令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集；
    if D_v 为空 **then**
      将 nxt 标记为叶结点，其 label 为 D_v 中样本最多的类；
      建边 $(cur, nxt, \text{label} = a_*)$ **return**
    else
      TreeGenerate(D_v, $A \setminus \{a_*\}$, nxt)
      建边 $(cur, nxt, \text{label} = a_*)$
    end if
  end for
end function

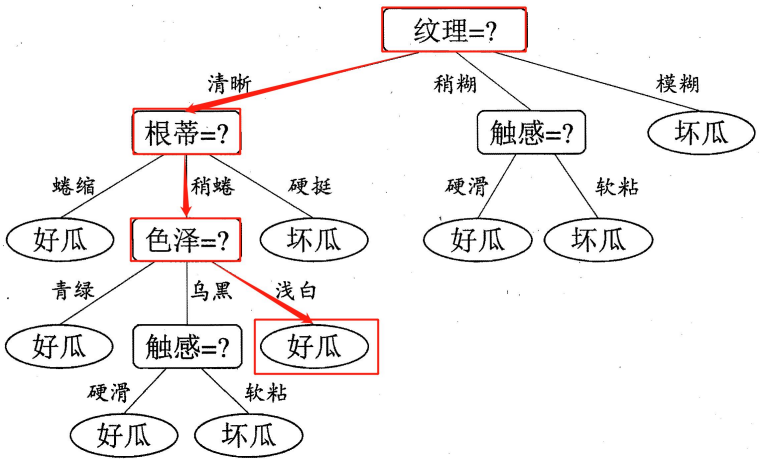

图 4: 样本预测举例

1.3.6 后剪枝

后剪枝在建树过后进行，自底向上标记深度最大的非叶结点为出现个数最多的类，并删除其叶结点。

为了做到自底向上删除结点，我们需要在建树后对其再进行一次 DFS，得到每个结点的深度，定

算法 3 决策树学习 BFS

输入： 训练集 $D_0 = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_m, y_m)\}$；初始属性集 $A_0 = \{a_1, a_2, \cdots, a_d\}$。

function TreeGenerate(D_0, A_0, cur_0)
  define 搜索队列 Q;
  初始化决策树，加入唯一的结点，索引为 1;
  push $(D_0, A_0, 1)$ **to** Q
  while Q 非空 **do**
    $(D, A, cur) \leftarrow$ **front of** Q;
    pop Q;
    if D 中样本全属于同一类别 C **then**
      将 cur 标记为 C 类叶结点，其 label 为 C；**continue**
    end if
    if $A \neq \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
      将 cur 标记为叶结点，其 label 标记为 D 中样本最多的类；**continue**
    end if
    从 A 中选择最优划分属性 a_*;
    将 cur 的 label 标记为 a_*;
    for a_* 的每个值 a_*^v **do**
      为 cur 生成一个分支结点 nxt；令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
      if D_v 为空 **then**
        将 nxt 标记为叶结点，其 label 为 D_v 中样本最多的类;
        建边 $(cur, nxt, \text{label} = a_*)$ **continue**
      else
        建边 $(cur, nxt, \text{label} = a_*)$
        push $(D_v, A \setminus \{a_*\}, nxt)$ **to** Q
      end if
    end for
  end while
end function

义结点 i 的深度为 d_i，根结点的深度为 1，则

$$d_{\text{son}_i} = d_i + 1 \tag{6}$$

得到深度数组后，将结点按照深度降序排列，依次队深度最大的非叶结点剪枝即可；整个后剪枝过程会依次删除深度最大的结点，但不影响其它结点的深度，因此深度数组无需在对一个结点进行一次剪枝后重新求。若对一个深度较大的结点进行剪枝后，能增大测试集的正确率，则对此结点剪枝。

算法 4 多变量决策树

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_m, y_m)\}$；属性集 $A = \{a_1, a_2, \cdots, a_d\}$；当前结点索引 cur。

function TreeGenerate(D, A, cur)
  if D 中样本全属于同一类别 C **then**
    将 cur 标记为 C 类叶结点，其 label 为 C；**return**
  end if
  使用对率回归进行二分类，得到两个子集 D_1 和 D_2
  if $D_1 = \varnothing$ **OR** $D_2 = \varnothing$ **then**
    将 cur 标记为叶结点，其 label 标记为 D 中样本最多的类；**return**
  end if
  生成结点 nxt_1 表示子集 D_1
  生成结点 nxt_2 表示子集 D_2
  建边 (cur, nxt_1)
  建边 (cur, nxt_2)
  TreeGenerate(D_1,nxt_1)
  TreeGenerate(D_2,nxt_2)
end function

1.4 核心代码详解

1.4.1 函数简介

函数名	功能
split_data_and_target(D)	将样本集 D 分割为属性集 X 和真实标记 y
Ent(D)	计算 D 的信息熵
get_Dv(D, a, v)	获得样本 D 中属性 a 取值为 v 的子集
Gain(D, a, class_dicts)	计算信息增益
Gini(D)	计算基尼值
Gini_index(D, a, class_dicts)	计算基尼系数
count_value_num(X, v)	矩阵 X 中包含行向量 v 的个数
get_greatest_split_attribute(D, A, solver='Gini_index')	选择最优划分属性

1.4.2 数据读取

在此次实验中，我使用了Lymphography、Balance Scale、Tic-Tac-Toe、Ionosphere四个 UCI 数据集，根据不同数据集的格式，我专门定义了读取数据的程序 DataLoader。

DataLoader 读取不同的数据集，并将数据集转换为统一的格式提供给数据集使用者。DataLoader 提供的数据集用 NumPy 的 ndarray 存储，每一行代表一个样本，最后一列为样本的真实值；DataLoader 还提供一个字典 class_dicts，字典的 key 为属性、value 为属性可取的值。

```
1 import numpy as np
```

```
2  import pandas as pd
3
4  def load(data_name):
5      if data_name == 'lymphography':
6          return load_lymphography()
7      elif data_name == 'balance-scale':
8          return load_balance_scale()
9      elif data_name == 'tic-tac-toe':
10         return load_tic_tac_toe()
11     elif data_name == 'ionosphere':
12         return load_ionosphere()
13     else:
14         raise Exception('数据源不存在')
15
16
17 def load_lymphography():
18     """
19     读取lymphography.data
20     第1列为分类，其余为属性值。
21     :return: 返回属性值，真实标记，每个属性可取的值。
22     """
23     lymphography = pd.read_csv('../data/lymphography.data', header=None)
24     X = lymphography.iloc[:, 1::].values
25     y = lymphography.iloc[:, 0].values
26     class_dicts = {}
27     for i in  range(18):
28         class_dicts[i] =  set(X[:, i])
29     D = np.c_[X, y]
30     return D, class_dicts
31
32
33 def load_balance_scale():
34     """
35     读取balance_scale.data
36     第1列为分类，其余为属性值。
37     :return: 属性值，真实标记，每个属性可取的值。
38     """
39     balance_scale = pd.read_csv('../data/balance-scale.data', header=None)
40     X = balance_scale.iloc[:, 1::].values
41     y = balance_scale.iloc[:, 0].values
42     class_dicts = {}
43     for i in  range(4):
44         class_dicts[i] =  set(X[:, i])
45     D = np.c_[X, y]
46     return D, class_dicts
47
```

```
48
49  def load_tic_tac_toe():
50      """
51      读取tic-tac-toe.data
52      最后一列为分类，其余为属性值
53      :return: 属性值，真实标记，每个属性可取的值。
54      """
55      tic_tac_toe = pd.read_csv('../data/tic-tac-toe.data', header=None)
56      X = tic_tac_toe.iloc[:, :9].values
57      y = tic_tac_toe.iloc[:, -1].values
58      class_dicts = {}
59      for i in  range(9):
60          class_dicts[i] =  set(X[:, i])
61      D = np.c_[X, y]
62      np.random.shuffle(D)
63      return D, class_dicts
64
65
66  def load_ionosphere():
67      """
68      读取ionosphere.data
69      最后一列为分类，其余为属性值
70      :return: 属性值，真实标记
71      """
72      ionosphere = pd.read_csv('../data/ionosphere.data', header=None)
73      X = ionosphere.iloc[:, :-1].values
74      y = ionosphere.iloc[:, -1].values
75      D = np.c_[X, y]
76      return D
```

1.4.3 决策树学习

依据算法2，未剪枝决策树学习的过程如下。代码中，叶结点的标记用“class: ”开头，表示类别；非叶结点的标记用“attribute: ”开头，表示划分属性；边上标记属性值。

```
1   def generate_normal_tree(D, A, cur_node_index):
2       """
3       建树
4       :param D: 训练集
5       :param A: 属性字典
6       :param cur_node_index: 当前点的索引
7       :return: void
8       """
9       X, y = split_data_and_target(D)
10      # 样本都属于同一类别
11      if  len( set(y)) == 1:
```

```
        g.nodes[cur_node_index]['label'] = 'class: {}'. format(y[0])
        return

    # 属性集为空或所有样本的取值都相同
    if  len(A) == 0 or count_value_num(X, X[0]) ==  len(X):
        # 找到出现次数最多的类别
        g.nodes[cur_node_index]['label'] = 'class: {}'.
            format(collections.Counter(y).most_common(1)[0][0])
        return
    # 选出最优属性
    attribute_index = get_greatest_split_attribute(D, A, solver='Gain')
    # 标记当前节点行为
    g.nodes[cur_node_index]['label'] = 'attribute: {}'. format(attribute_index)
    # 对最优属性的每个取值产生一个分支
    for v in A[attribute_index]:
        # 所有数据中在最优属性上取值为v的子集
        Dv = get_Dv(D, attribute_index, v)
        node_num = g.number_of_nodes()
        if  len(Dv) == 0:
            # 创建分支节点
            # 将分支节点标记为叶节点，其类别为D中样本最多的类别
            g.add_node(node_num + 1, label='class: {}'.
                format(collections.Counter(y).most_common(1)[0][0]))
            g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
        else:
            # 创建分支节点
            # 分支节点的属性选择需要下一步递归确定
            g.add_node(node_num + 1, label=None)
            g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
            new_A = copy.deepcopy(A)
            new_A.pop(attribute_index)
            generate_normal_tree(Dv, new_A, node_num + 1)
```

1.4.4 多变量决策树

依据算法4，基于对率回归的多变量决策树学习过程如下。代码中，直接将样本集合拆分为参数矩阵 X 和真实标记向量 y 传入。叶结点的标记用“class: ”开头，表示类别；非叶结点的标记用于表示对率回归分类器的参数值 $\boldsymbol{w}$ 和 b；基于对率回归的“二分类”特性，每个非叶结点产生两个分支，分别为 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b>0$ 和 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b\leqslant 0$。

```
def generate_normal_tree(X, y, cur_node_index):
    """
    建树
    :param D: 训练集
    :param A: 属性字典
    :param cur_node_index: 当前点的索引
    :return: void
```

```
    """
    # 样本都属于同一类别
    if  len( set(y)) == 1:
        g.nodes[cur_node_index]['label'] = 'class{}'. format(y[0])
        return

    # 进行线性分类
    clf = LogisticRegression().fit(X, y)
    w, b = clf.coef_[0], clf.intercept_[0]
    g.nodes[cur_node_index]['w'] = w
    g.nodes[cur_node_index]['b'] = b
    g.nodes[cur_node_index]['label'] = 'w={},b={}'. format(w, b)
    # 得到分类后的子集
    X1, y1 = [], []
    X2, y2 = [], []
    for i in  range( len(y)):
        if np.dot(w, X[i]) + b > 0:
            X1.append(X[i])
            y1.append(y[i])
        else:
            X2.append(X[i])
            y2.append(y[i])

    X1 = np.array(X1)
    X2 = np.array(X2)
    y1 = np.array(y1)
    y2 = np.array(y2)
    if  len(y1) == 0 or  len(y2) == 0:
        # 不可分类
        g.nodes[cur_node_index]['label'] = 'class: {}'.
            format(collections.Counter(y).most_common(1)[0][0])
        return

    # 产生分支节点
    node_num = g.number_of_nodes()
    g.add_node(node_num + 1, label=None, w=None, b=None)
    g.add_node(node_num + 2, label=None, w=None, b=None)
    g.add_edge(cur_node_index, node_num + 1, label='wx+b>0')
    g.add_edge(cur_node_index, node_num + 2, label='wx+b<=0')
    generate_normal_tree(X1, y1, node_num + 1)
    generate_normal_tree(X2, y2, node_num + 2)
```

1.4.5 预剪枝

预剪枝在建树过程中进行，这里使用 BFS 实现。在算法3的基础上，建树过程增加了准确性的判断，仅当继续产生分支结点得到的决策树准确率更高，才继续搜索过程。

```python
def generate_pre_pruning_tree(ini_D, ini_A):
    """
    预剪枝建树
    :param ini_D: 训练集
    :param ini_A: 属性字典
    :return: void
    """
    ini_X, ini_y = split_data_and_target(ini_D)
    g.add_node(1, label='class: {}'. format(collections.Counter(ini_y).most_common(1)[0][0]))
    q = Queue()
    # 初始状态入队
    q.put([1, ini_D, ini_A])
    while not q.empty():
        cur = q.get_nowait()
        # 取队头
        cur_node_index, D, A = cur[0], cur[1], cur[2]
        X, y = split_data_and_target(D)
        attribute_index = get_greatest_split_attribute(D, A, solver='Gain')

        # 停止搜索得到的决策树
        stop_g = copy.deepcopy(g)
        stop_g.nodes[cur_node_index]['label'] = 'class: {}'.
            format(collections.Counter(y).most_common(1)[0][0])

        # 产生分支结点的决策树
        divide_g = copy.deepcopy(g)
        divide_g.nodes[cur_node_index]['label'] = 'attribute: {}'. format(attribute_index)
        # 遍历每一个可能取值v生成子结点
        for v in class_dicts[attribute_index]:
            Dv = get_Dv(D, attribute_index, v)
            node_num = divide_g.number_of_nodes()
            if  len(Dv) == 0:
                divide_g.add_node(node_num + 1, label='class: {}'.
                    format(collections.Counter(y).most_common(1)[0][0]))
                divide_g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
            else:
                Xv, yv = split_data_and_target(Dv)
                divide_g.add_node(node_num + 1, label='class: {}'.
                    format(collections.Counter(yv).most_common(1)[0][0]))
                divide_g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))

        # 产生分支的决策树准确率低
        if get_accuracy(stop_g, D_test) >= get_accuracy(divide_g, D_test):
            continue

        # 产生分支
        g.nodes[cur_node_index]['label'] = 'attribute: {}'. format(attribute_index)
```

```
        for v in class_dicts[attribute_index]:
            Dv = get_Dv(D, attribute_index, v)
            node_num = g.number_of_nodes()
            if  len(Dv) == 0:
                g.add_node(node_num + 1, label='class: {}'.
                    format(collections.Counter(y).most_common(1)[0][0]))
                g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
            else:
                Xv, yv = split_data_and_target(Dv)
                g.add_node(node_num + 1, label='class: {}'.
                    format(collections.Counter(yv).most_common(1)[0][0]))
                g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
                new_A = copy.deepcopy(A)
                new_A.pop(attribute_index)
                q.put([node_num + 1, Dv, new_A])
```

1.4.6 后剪枝

后剪枝在建树过后进行。首先需要调用 get_depth 函数获得每个结点的深度，然后从深度大的非叶结点开始，依次对非叶结点进行剪枝。

```
def get_depth(cur):
    """
    获取结点深度
    :param cur: 当前结点索引
    :return: void
    """
    # depth[i][0] 为深度
    # depth[i][1] 为结点索引
    for nxt in g.neighbors(cur):
        depth[nxt][0] = depth[cur][0] + 1
        depth[nxt][1] = nxt
        get_depth(nxt)

# 按结点深度降序
depth = np.array( sorted(depth, key=lambda d: d[0], reverse=True), dtype= int)

def generate_post_pruning_tree(g):
    """
    后剪枝
    :param g: 需要剪枝的决策树
    :return: void
    """
    # 按照结点深度降序遍历
    for node in depth:
        # 遇到叶结点跳过
        if g.nodes[node[1]]['label'].startswith('class: '):
```

```
            continue

        # 考量去掉其子结点后准确率是否提高
        new_g = copy.deepcopy(g)
        for nei in  list(new_g.neighbors(node[1])):
            new_g.remove_node(nei)
        Dv = new_g.nodes[node[1]]['data']
        X, y = split_data_and_target(Dv)
        new_g.nodes[node[1]]['label'] = 'class: {}'. format(collections.Counter(y).most_common(1)[0][0])
        # 剪枝后准确率更高
        if get_accuracy(new_g, D_test) >= get_accuracy(g, D_test):
            g = new_g
    return g
```

1.5 实验结果分析

使用Lymphography、Balance Scale、Tic-Tac-Toe三个 UCI 数据集对 ID3 算法和 CART 算法进行测试。

针对每个数据集训练一棵决策树，取数据集的 80% 作为训练集，其余作为测试集，测试集的准确率如表1和表2。可以发现，只有 Tic-Tac-Toe 数据集，预剪枝后的准确率较未剪枝的决策树低，其余的剪枝操作都能使得预测结果提高。对比图6和图7可看出，Tic-Tac-Toe 数据集的后剪枝决策树相比预剪枝决策树保留了更多的分支。可初步得出结论：后剪枝决策树的欠拟合风险小，泛化性能优于预剪枝决策树。由于后剪枝过程是在完成决策树之后进行的，并且要自底向上地对树中的所有非叶结点进行逐一考查，因此其训练时间开销比未剪枝决策树和预剪枝决策树都大得多。

	未剪枝	预剪枝	后剪枝
Lymphography	0.667	0.767	0.767
Balance Scale	0.664	0.704	0.704
Tic-Tac-Toe	0.875	0.703	0.943

表 1: ID3 结果

	未剪枝	预剪枝	后剪枝
Lymphography	0.633	0.767	0.767
Balance Scale	0.664	0.704	0.704
Tic-Tac-Toe	0.875	0.703	0.943

表 2: CART 结果

以 Tic-Tac-Toe 数据集为例，图5为未剪枝决策树，图6为预剪枝的决策树，图7为后剪枝的决策树。可以发现，剪枝后的决策树不仅结点数和深度更小，能够提高样本的预测速度；基于表1和2，其准确率也会提高。

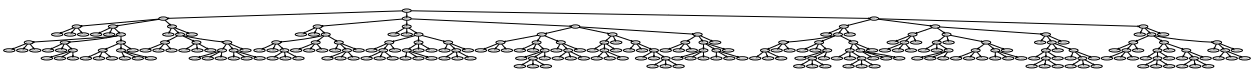

图 5: Tic-Tac-Toe 数据集未剪枝 ID3 决策树

使用Ionosphere数据集对基于对率回归的多变量决策树进行测试，其准确率为 0.944。未剪枝多变量决策树准确率就已经足够高，且决策树的结点数和深度也不大。多变量决策树的优点在于，其斜向的分类边界能够加快分类速度，提高分类准确率。

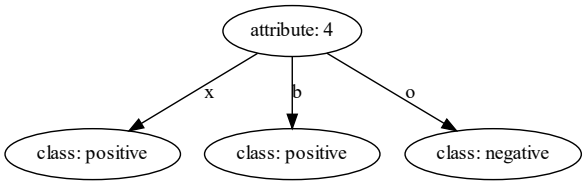

图 6: Tic-Tac-Toe 数据集预剪枝 ID3 决策树

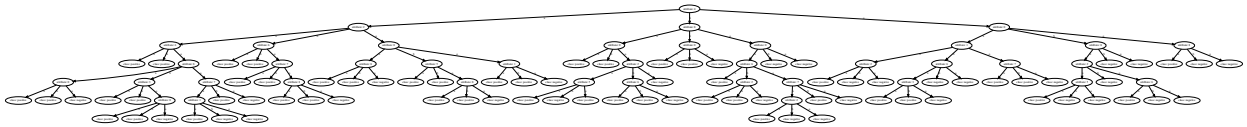

图 7: Tic-Tac-Toe 数据集后剪枝 ID3 决策树

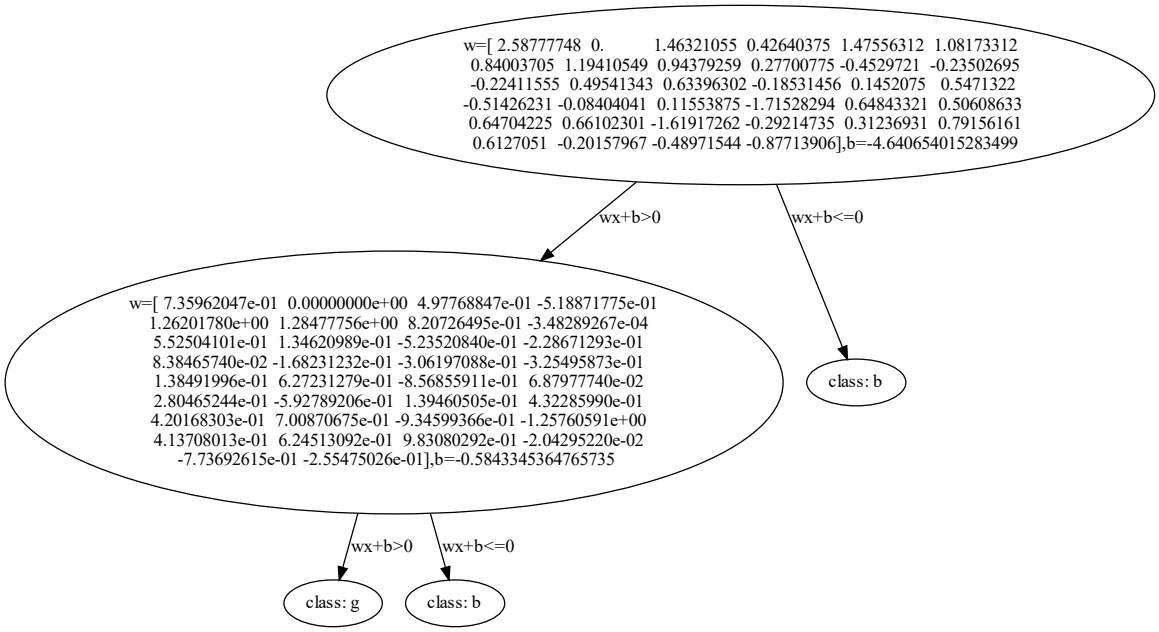

图 8: Ionosphere 数据集基于对率回归的决策树

1.6 学习收获

在此次实验中，我尝试实现了多个决策树学习算法，包括 ID3、CART、多变量决策树等，并对决策树进行了剪枝操作，得出了“后剪枝较预剪枝”更优的结论。此次实验只利用决策树完成了分类任务，还可以进一步使用决策树完成回归任务。个人认为，决策树逻辑简单、易实现、没有很强的数学性，但

是所占的空间复杂度有一些大。

相比自己实现的决策树学习算法，sklearn 提供的决策树学习算法处理了连续值和离散值，可控制决策树生长时的各类阈值，同时能借助 graphviz 画出决策树。如图9，是利用 sklearn 生成的 iris 数据集决策树。

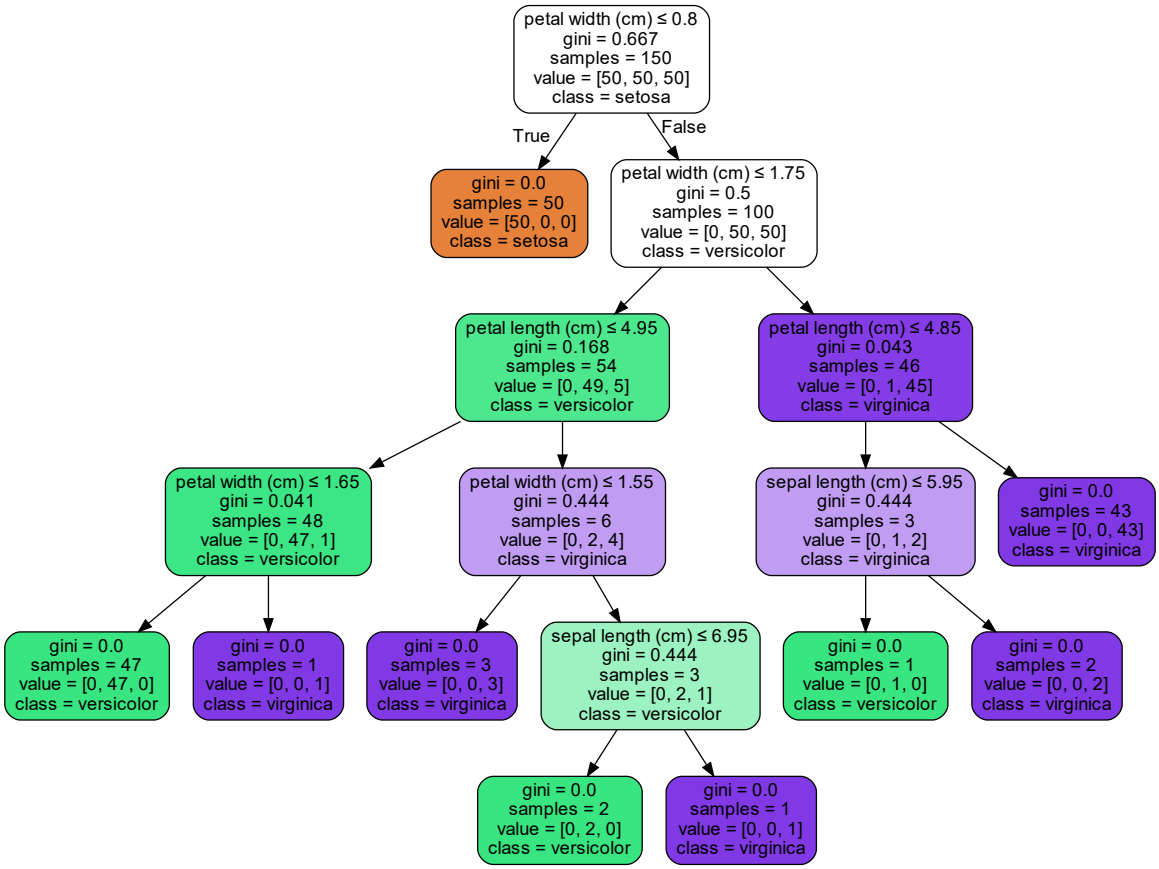

图 9: iris 数据集决策树

1.7 参考资料

- 决策树分类（scikit-learn 中文社区）
- sklearn.tree.DecisionTreeClassifier（sckikit-learn 中文社区）
- 《机器学习》4. 决策树；清华大学出版社；周志华

2 习题 2

2.1 编程题目理解

题目要求：使用”队列”数据结构，以参数 MaxDepth 控制树的最大深度，写出使用非递归的决策树生成算法。算法3已经给出了非递归的，即广度优先的建树算法，对此加以改造即可。

2.2 算法设计思路

定义状态 $(cur, D, A, depth)$，分别表示当前结点编号、样本集合、属性集合、结点深度。相比算法3，这里向 BFS 记录的状态中添加了“深度”这一状态。每次生成分支节点，都将新结点的深度在当前结点的基础上增加 1，将新的结点加入队列。当结点的深度达到 MaxDepth 时，则停止搜索。

2.3 核心代码分析

在建树过程中，记录结点的深度，当深度达到 MaxDepth 则停止搜索。

```
def generate_normal_tree(ini_D, ini_A, MaxDepth):
    """
    建树
    :param D: 训练集
    :param A: 属性字典
    :param cur_node_index: 当前点的索引
    :return: void
    """
    ini_X, ini_y = split_data_and_target(ini_D)
    g.add_node(1, label='class: {}'. format(collections.Counter(ini_y).most_common(1)[0][0]))
    q = Queue()
    q.put([1, ini_D, ini_A, 1])
    while not q.empty():
        cur = q.get_nowait()
        cur_node_index, D, A, node_depth = cur[0], cur[1], cur[2], cur[3]
        if node_depth == MaxDepth:
            continue

        X, y = split_data_and_target(D)

        # 样本都属于同一类别
        if  len( set(y)) == 1:
            g.nodes[cur_node_index]['label'] = 'class: {}'. format(y[0])
            continue

        # 属性集为空或所有样本的取值都相同
        if  len(A) == 0 or count_value_num(X, X[0]) ==  len(X):
            # 找到出现次数最多的类别
            g.nodes[cur_node_index]['label'] = 'class: {}'.
                format(collections.Counter(y).most_common(1)[0][0])
```

```
30            continue
31
32        attribute_index = get_greatest_split_attribute(D, A, solver='Gain')
33
34        g.nodes[cur_node_index]['label'] = 'attribute: {}'. format(attribute_index)
35        for v in A[attribute_index]:
36            Dv = get_Dv(D, attribute_index, v)
37            node_num = g.number_of_nodes()
38            if  len(Dv) == 0:
39                g.add_node(node_num + 1, label='class: {}'.
                      format(collections.Counter(y).most_common(1)[0][0]))
40                g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
41            else:
42                Xv, yv = split_data_and_target(Dv)
43                g.add_node(node_num + 1, label='class: {}'.
                      format(collections.Counter(yv).most_common(1)[0][0]))
44                g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
45                new_A = copy.deepcopy(A)
46                new_A.pop(attribute_index)
47                q.put([node_num + 1, Dv, new_A, node_depth + 1])
```

2.4 实验结果

使用西瓜数据集 2.0 学习得到的决策树如图10，其准确率为 0.4286。

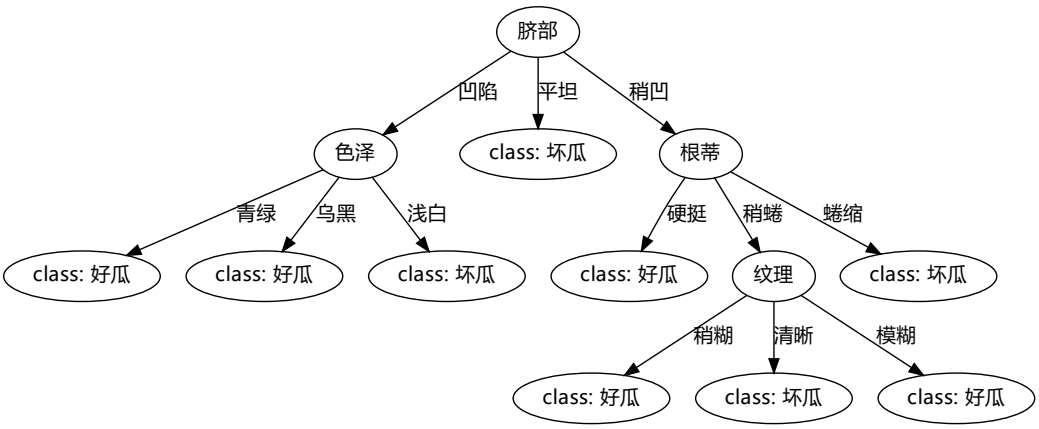

图 10: 限制结点深度得到的决策树

2.5 学习收获

构建决策树时限制结点的最大深度，能够有效减少递归的层数、加快决策树的学习速度，也避免了一些过拟合的风险；但是这本质上还是一种类似预剪枝的策略，且没有剪枝时的准确率检验。

3 习题 3

3.1 编程题目理解

题目要求：以参数 MaxNode 控制树的最大结点数，写出使用非递归的决策树生成算法。算法3已经给出了非递归的，即广度优先的建树算法，对此加以改造即可。

3.2 算法设计思路

定义状态 (cur, D, A)，分别表示当前结点编号、样本集合、属性集合，限制条件“结点数”由 NetworkX 维护。每次生成分支节点，都考虑生成分支结点后决策树结点是否会超过 MaxNode；若不超过，则生成分支结点；否则，标记当前结点为样本中出现最多的种类，并停止搜索。

3.3 核心代码

在建树过程中，维护决策树的结点数，当扩展结点后结点数超过 MaxNode 则停止搜索。

```
def generate_normal_tree(ini_D, ini_A, MaxNode):
    """
    建树
    :param D: 训练集
    :param A: 属性字典
    :param cur_node_index: 当前点的索引
    :return: void
    """
    ini_X, ini_y = split_data_and_target(ini_D)
    g.add_node(1, label='class: {}'. format(collections.Counter(ini_y).most_common(1)[0][0]))
    q = Queue()
    q.put([1, ini_D, ini_A])
    while not q.empty():
        cur = q.get_nowait()
        cur_node_index, D, A = cur[0], cur[1], cur[2]

        X, y = split_data_and_target(D)

        # 样本都属于同一类别
        if  len( set(y)) == 1:
            g.nodes[cur_node_index]['label'] = 'class: {}'. format(y[0])
            continue

        # 属性集为空或所有样本的取值都相同
        if  len(A) == 0 or count_value_num(X, X[0]) ==  len(X):
            # 找到出现次数最多的类别
            g.nodes[cur_node_index]['label'] = 'class: {}'.
                format(collections.Counter(y).most_common(1)[0][0])
            continue

```

```
30          attribute_index = get_greatest_split_attribute(D, A, solver='Gain')
31
32          # 超过限制则不扩展
33          if g.number_of_nodes() +  len(A[attribute_index]) > MaxNode:
34              return
35
36          g.nodes[cur_node_index]['label'] = 'attribute: {}'. format(attribute_index)
37
38          for v in A[attribute_index]:
39              Dv = get_Dv(D, attribute_index, v)
40              node_num = g.number_of_nodes()
41              if  len(Dv) == 0:
42                  g.add_node(node_num + 1, label='class: {}'.
                        format(collections.Counter(y).most_common(1)[0][0]))
43                  g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
44              else:
45                  Xv, yv = split_data_and_target(Dv)
46                  g.add_node(node_num + 1, label='class: {}'.
                        format(collections.Counter(yv).most_common(1)[0][0]))
47                  g.add_edge(cur_node_index, node_num + 1, label='{}'. format(v))
48                  new_A = copy.deepcopy(A)
49                  new_A.pop(attribute_index)
50                  q.put([node_num + 1, Dv, new_A])
```

3.4 实验结果分析

使用西瓜数据集 2.0 学习得到的决策树如图11，其准确率为 0.5714。

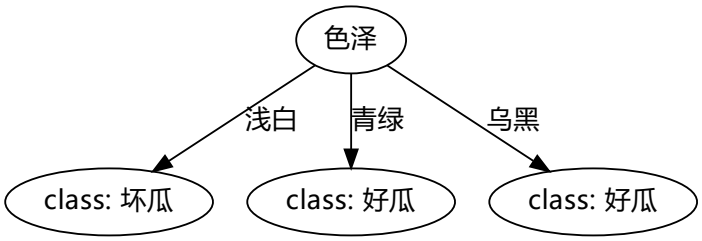

图 11: 限制结点数得到的决策树

3.5 学习收获

限制 MaxDepth 能有效减小树的深度，在预测样本值时能更快到达叶结点。限制 MaxNode 能有效减少决策树的空间复杂度。

由于生成分支结点时，需要将一个属性的所有分支结点一次性生成，否则在预测样本时缺失属性值而导致无法到达叶结点；因此 MaxNode 取值不当时，可能出现生成分支结点后决策树结点数仅仅只超出 MaxNode 一些，使得当前结点无法继续搜索的情况。针对这种情况，不妨对 MaxNode 设置一个容错空间 ϵ，即决策树可超出限制 ϵ 个结点。