第十三周学习笔记—决策树剪枝

2022-07-02

第一章　CART 决策树

1.1　CART 决策树原理

决策树 (decision tree) 是一类常见的机器学习方法。在已知各种情况发生概率的基础上，通过构成决策树来求取净现值的期望值大于等于零的概率，评价项目风险，判断其可行性的决策分析方法，是直观运用概率分析的一种图解法。在机器学习中，决策树是一个预测模型，他代表的是对象属性与对象值之间的一种映射关系。

CART（Classification And Regression Trees 分类回归树）算法是一种树构建算法，既可以用于分类任务，又可以用于回归。相比于 ID3 和 C4.5 只能用于离散型数据且只能用于分类任务，CART 算法的适用面要广得多，既可用于离散型数据，又可以处理连续型数据，并且分类和回归任务都能处理。

本次仅讨论基本的 CART 分类决策树构建和剪枝问题，不讨论回归树等问题。

CART 决策树使用基尼指数来选择划分属性，这是西瓜书上给出的定义。我们将 CART 决策树关键点整理如下：

（1）CART 决策树既能是分类树，也能是回归树

（2）当 CART 是分类树时，采用 GINI 值作为节点分裂的依据

（3）CART 是一颗二叉树 (关于这一点其实我存在疑惑。因为在我编程实现的过程中忽略了这一点，西瓜树和一些课程上并没有明确指明 CART 算法必须生成二叉树，大多数也和 ID3 算法等生成的树类，生成的是多叉树。所以是否考虑在属性取值较少的情况下，CART 算法不用一定生成二叉树)

1.2　选择划分属性

CART 决策树作为分类树时，特征属性可以是连续类型也可以是离散类型，但观察属性 (即标签属性或者分类属性) 必须是离散类型。

划分的目的是为了能够让数据变纯，使决策树输出的结果更接近真实值。

如果是分类树，CART 决策树使用“基尼指数”来选择划分属性，数据集 D 的纯度可以用基尼值来度量。

具体的计算公式本次不再重复。

1.3　CART 决策树生成算法

定理. 输入：训练数据集 D，停止计算的条件

输出：*CART* 决策树

根据训练数据集，从根结点开始，递归地对每个结点进行以下操作，构建二叉决策树：

（*1*）计算现有特征对该数据集的基尼指数；

（*2*）选择基尼指数最小的值对应的特征为最优特征，对应的切分点为最优切分点（若最小值对应的特征或切分点有多个，随便取一个即可）；

（*3*）按照最优特征和最优切分点，从现结点生成两个子结点，将训练数据集中的数据按特征和属性分配到两个子结点中；

（*4*）对两个子结点递归地调用（*1*）（*2*）（*3*），直至满足停止条件。

（*5*）生成 *CART* 树。

算法停止的条件：结点中的样本个数小于预定阈值，或样本集的基尼指数小于预定阈值（样本基本属于同一类，如完全属于同一类则为 *0*），或者特征集为空。

注：最优切分点是将当前样本下分为两类（因为我们要构造二叉树）的必要条件。对于离散的情况，最优切分点是当前最优特征的某个取值；对于连续的情况，最优切分点可以是某个具体的数值。具体应用时需要遍历所有可能的最优切分点取值去找到我们需要的最优切分点。

1.4　CART 算法的 Python 实现

1.4.1　多叉树形式

创建数据集

```
def create_data():
    with open('./2022-06-19data/data_word.csv',encoding='utf-8',newline='\n
                                            ') as f:
        reader=csv.reader(f) #此处读取到的数据是将每行数据当做列表返回的
```

```
        data=[]
        for row in reader: #此时输出的是一行行的列表
            data.append(row)
    features=data[0][:-1]
    dataset=data[1:]
    return dataset,features
```

我们展示一下数据集的格式：

```
数据集为:
 [['青绿', '蜷缩', '浊响', '清晰', '凹陷', '硬滑', '1'], ['乌黑', '蜷缩', '
                                                沉闷', '清晰', '凹陷', '硬滑', '1'],
                                                 ['乌黑', '蜷缩', '浊响', '清晰', '
                                                凹陷', '硬滑', '1'], ['青绿', '蜷缩'
                                                , '沉闷', '清晰', '凹陷', '硬滑', '1
                                                '], ['浅白', '蜷缩', '浊响', '清晰',
                                                 '凹陷', '硬滑', '1'], ['青绿', '稍
                                                蜷', '浊响', '清晰', '稍凹', '软粘',
                                                 '1'], ['乌黑', '稍蜷', '浊响', '稍
                                                糊', '稍凹', '软粘', '1'], ['乌黑',
                                                '稍蜷', '浊响', '清晰', '稍凹', '硬
                                                滑', '1'], ['乌黑', '稍蜷', '沉闷',
                                                '稍糊', '稍凹', '硬滑', '0'], ['青绿
                                                ', '硬挺', '清脆', '清晰', '平坦', '
                                                软粘', '0'], ['浅白', '硬挺', '清脆'
                                                , '模糊', '平坦', '硬滑', '0'], ['浅
                                                白', '蜷缩', '浊响', '模糊', '平坦',
                                                 '软粘', '0'], ['青绿', '稍蜷', '浊
                                                响', '稍糊', '凹陷', '硬滑', '0'], [
                                                '浅白', '稍蜷', '沉闷', '稍糊', '凹
                                                陷', '硬滑', '0'], ['乌黑', '稍蜷',
                                                '浊响', '清晰', '稍凹', '软粘', '0']
                                                , ['浅白', '蜷缩', '浊响', '模糊', '
                                                平坦', '硬滑', '0'], ['青绿', '蜷缩'
                                                , '沉闷', '稍糊', '稍凹', '硬滑', '0
                                                ']]
数据标签为:
 ['色泽', '根蒂', '敲声', '纹理', '脐部', '触感']
```

同样的，我们可以换另一种表达形式:

```
def create_data():
    df=pd.read_csv('./2022-06-19data/data_word.csv')
    data=df.values[:,:].tolist()
    labels = df.columns.values[:-1].tolist()
    return data,labels
```

最后我们得到的结果与上面结果相同。

计算当前集合的 Gini 系数

```
def CART_Gini(dataset):
    # 计算参与训练的数据量，即训练集中共有多少条数据;
    num_examples = len(dataset)
    # 计算每个分类标签label在当前节点出现的次数，即每个类别在当前节点下别分
                                                别出现多少次，用作信息熵概率 p 的
                                                分子
    label_counts = {}
    # 每次读入一条样本数据
    for feature_vector in dataset:
        # 将当前实例的类别存储，即每一行数据的最后一个数据代表的是类别
        current_label = feature_vector[-1]
        # 为所有分类创建字典，每个键值对都记录了当前类别出现的次数
        if current_label not in label_counts:
            label_counts[current_label] = 0
        label_counts[current_label] += 1
    # 使用循环方法，依次求出公式求和部分每个类别所占的比例及相应的计算
    gini = 1  # 记录基尼值
    for key in label_counts:
        # 算出该类别在该节点数据集中所占的比例
        prob = label_counts[key] / num_examples
        gini -= prob * prob
    return gini
```

提取子集合

从 dataset 中先找到所有第 axis 个标签值 = value 的样本, 然后将这些样本删去第 axis 个标签值，再全部提取出来成为一个新的样本集。

```
def split_data(dataset, axis, value):
    # 存储划分好的数据集
```

```
    reduced_dataset = []
    for feature_vector in dataset:
        if feature_vector[axis] == value:
            reduced_feature_vector = feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis + 1:])
            reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset
```

选择最佳的划分特征

```
def choose_best_feature_to_split(dataset):
    # 求该数据集中共有多少特征（由于最后一列为label标签，所以减1）
    num_features = len(dataset[0]) - 1
    # 初始化最优基尼指数，和最优的特征索引
    best_gini_index, best_feature = 1, -1
    # 依次遍历每一个特征，计算其基尼指数
    # 当前数据集中有几列就说明有几个特征
    for i in range(num_features):
        # 将数据集中每一个特征下的所有属性拿出来生成一个列表
        feature_list = [example[i] for example in dataset]
        # 使用集合对列表进行去重，获得去重后的集合，即：此时集合中每个属性
                                                只出现一次
        unique_values = set(feature_list)
        # 创建一个临时基尼指数
        new_gini_index = 0
        # 依次遍历该特征下的所有属性
        for value in unique_values:
            # 依据每一个属性划分数据集
            sub_dataset = split_data(dataset, i, value)  # 详情见
                                                    splitDataSet函数
            # 计算该属性包含的数据占该特征包含数据的比例
            prob = len(sub_dataset) / len(dataset)
            # 计算每个属性结点的基尼值，并乘以相应的权重，再求他们的和，即
                                                    为该特征节点的基尼指数
            new_gini_index += prob * CART_Gini(sub_dataset)
            # 比较所有特征中的基尼指数，返回最好特征划分的索引值，注意：基
                                                    尼指数越小越好
        if (new_gini_index < best_gini_index):
            best_gini_index = new_gini_index
```

```
            best_feature = i
            # 返回最优的特征索引
    return best_feature
```

返回出现次数最多的类别

```
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1  # classCount以字典形式记录每个类别出现的次
                                数
    # 倒叙排列classCount得到一个字典集合，然后取出第一个就是结果（好瓜/坏
      瓜），即出现次数最多的结果
    # sortedClassCount的形式是[(),(),(),...],每个键值对变为元组并以列表形式
      返回
    sorted_class_count = sorted(class_count.items(), key=operator.
                                itemgetter(1), reverse=True)
    return sorted_class_count[0][0]  # 返回的是出现类别次数最多的“类别”
```

创建决策树

```
def create_tree(dataset, labels):
    # classList中记录了该节点数据中“类别”一列，保存为列表形式
    class_list = [example[-1] for example in dataset]
    # 如果该结点为空集，则将其设为叶子结点，节点类型为其父节点中类别最多的
      类。
    if len(dataset) == 0:
        return majority_cnt(class_list)
    # 第一个停止条件：所有的类标签完全相同，则直接返回该类标签
    # 如果数据集到最后一列的第一个值出现的次数=整个集合的数量，也就说只有一
      个类别，直接返回结果即可
    if class_list.count(class_list[0]) == len(class_list):  # count()函数是
      统计括号中的值在list中出现的次数
        return class_list[0]
    # 第二个停止条件：使用完了所有特征，仍然不能将数据集划分成仅包含唯一类
      别的分组
```

```
    # 如果最后只剩一个特征，那么出现相同label多的一类，作为结果
    if len(dataset[0]) == 1:  # 所有的特征都用完了，只剩下最后的标签列了
        return majority_cnt(class_list)
    # 选择最优的特征
    best_feature = choose_best_feature_to_split(dataset)  # 返回的是最优特
                                                征的索引
    # 获取最优特征
    best_feature_label = labels[best_feature]
    # 初始化决策树
    my_tree = {best_feature_label: {}}
    # 将使用过的特征数据删除
    del (labels[best_feature])
    # 在数据集中去除最优特征列，然后用最优特征的分支继续生成决策树
    feature_values = [example[best_feature] for example in dataset]
    unique_values = set(feature_values)
    # 遍历该特征下每个属性节点，继续生成决策树
    for value in unique_values:
        # 求出剩余的可用的特征
        sub_labels = labels[:]
        my_tree[best_feature_label][value] = create_tree(split_data(dataset
                                                , best_feature, value),
                                                sub_labels)
    return my_tree
```

决策树可视化

下面的代码就是直接借鉴的《机器学习实战》的代码，能够根据固定形式的字典生成决策树，可以直接使用。

详细的代码解释如下:

```
#绘图相关参数的设置
def plot_node(node_txt, center_pt, parent_pt, node_type):
    # annotate函数是为绘制图上指定的数据点xy添加一个nodeTxt注释
    # nodeTxt是给数据点xy添加一个注释，xy为数据点的开始绘制的坐标,位于节点
                                                的中间位置
    # xycoords设置指定点xy的坐标类型，xytext为注释的中间点坐标，textcoords
                                                设置注释点坐标样式
    # bbox设置装注释盒子的样式,arrowprops设置箭头的样式
    '''
```

```
    figure points:表示坐标原点在图的左下角的数据点
    figure pixels:表示坐标原点在图的左下角的像素点
    figure fraction: 此时取值是小数, 范围是([0,1],[0,1]),在图的左下角时xy是
                                        (0,0), 最右上角是(1,1)
    其他位置是按相对图的宽高的比例取最小值
    axes points : 表示坐标原点在图中坐标的左下角的数据点
    axes pixels : 表示坐标原点在图中坐标的左下角的像素点
    axes fraction : 与figure fraction类似, 只不过相对于图的位置改成是相对于
                                        坐标轴的位置
    '''
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                        fraction',xytext=center_pt,
                                        textcoords='axes fraction',va="
                                        center", ha="center", bbox=
                                        node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化树的叶子节点个数
    # python3中my——tree.keys()返回的是dict_keys,不在是list,所以不能使用
                                        myTree.keys()[0]的方法获取结点属
                                        性, 可以使用list(myTree.keys())[0
                                        ]
    first_str = list(my_tree.keys())[0]
    # 通过键名获取与之对应的值
    second_dict = my_tree[first_str]
    # 遍历树, secondDict.keys()获取所有的键
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                        典, 如果不是字典, 代表此结点
                                        为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
```

```
        else: #如果不是字典，则叶子结点的数目就加1
            num_leafs += 1
    return num_leafs #返回叶子节点的数目

"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0] #获取树的第一个键名
    second_dict = my_tree[first_str] #获取下一个字典,获取键名所对应的值
    for key in second_dict.keys(): #遍历树
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1 #如果获取的键
                                                             是字典，树的深度加1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #去深度的最大值
            max_depth = thisDepth
    return max_depth #返回树的深度

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
#绘制线中间的文字(0和1)的绘制
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0] #计算文字的x坐标
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1] #计算文字的y坐标
```

```
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""
#设置画节点用的盒子的样式
leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")

#绘制树
def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree) #获取树的叶子节点
    depth = get_tree_depth(my_tree) #获取树的深度
    first_str = list(my_tree.keys())[0] #获取第一个键名
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                        .total_w, plot_tree.y_off) #计算
                                        子节点的坐标
    plot_mid_text(cntr_pt, parent_pt, node_txt) #绘制线上的文字
    plot_node(first_str, cntr_pt, parent_pt, decision_node) #绘制节点
    second_dict = my_tree[first_str] #获取第一个键值
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d #计算节点y
                                        方向上的偏移量, 根据树的深度
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key)) #递归绘制树
        else:
            # 更新x的偏移量,每个叶子结点x轴方向上的距离为 1/plotTree.totalW
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            # 绘制非叶子节点
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                cntr_pt, leaf_node)
            # 绘制箭头上的标志
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
```

```
                                                    key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""
mpl.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white') #新建一个figure设
                                                    置背景颜色为白色
    fig.clf() #清除figure
    axprops = dict(xticks=[], yticks=[])
    # 创建一个1行1列1个figure, 并把网格里面的第一个figure的Axes实例返回给
                                                    ax1作为函数createPlot()
    # 的属性, 这个属性ax1相当于一个全局变量, 可以给plotNode函数使用
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree)) #获取树的叶子节点
    plot_tree.total_d = float(get_tree_depth(in_tree)) #获取树的深度
    #节点的x轴的偏移量为-1/plotTree.totlaW/2,1为x轴的长度, 除以2保证每一个
                                                    节点的x轴之间的距离为1/plotTree.
                                                    totlaW*2
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()
```

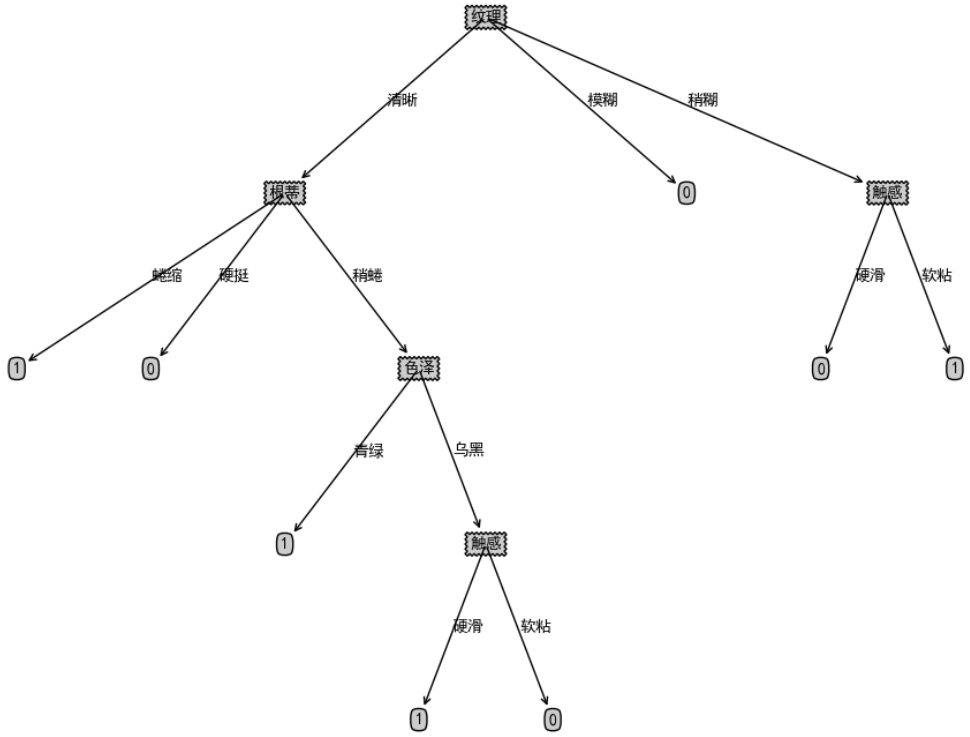
决策树的结果展示

根据上面的代码，我们可以得到最终的决策树为:

```
{'纹理': {'清晰': {'根蒂': {'稍蜷': {'色泽': {'乌黑': {'触感': {'硬滑': '1'
                                                    , '软粘': '0'}}, '青绿': '1'}}, '蜷缩
                                                    ': '1', '硬挺': '0'}}, '模糊': '0', '
```

```
                                    '稍糊': {'触感': {'硬滑': '0', '软粘':
                                    '1'}}}}
```

决策树可视化为:



决策树分类

上面我们实现了决策分类问题，接下来我们来定义决策树分类函数:

```
def classify(my_tree,test_data):
    first_feature=list(my_tree.keys())[0] #获取根节点
    second_dict=my_tree[first_feature] #获取下一级分支
    #查找当前列表中第一个匹配first_feature变量的元素的索引
    input_first=test_data.get(first_feature)
    input_value=second_dict[input_first] #获取测试样本通过第一个特征分类器
                                         后的输出
    if isinstance(input_value,dict): #判断结点是否为字典来判断是否为根节点
        class_label=classify(input_value,test_data)
    else:
        class_label=input_value #如果到达叶子结点，则返回当前节点的分类标签
```

```
    return class_label
```

我们拿一组数据集进行测试如下:

```
test_data_1 = {'色泽': '青绿', '根蒂': '蜷缩', '敲声': '浊响', '纹理': '清
                                        晰', '脐部': '平坦', '触感': '硬滑'}
result=classify(my_tree,test_data_1)
print('分类结果为'+'好瓜'if result=='1' else '分类结果为'+'坏瓜')
```

我们的预测结果如下:

```
分类结果为好瓜
```

1.4.2　二叉树形式

划分数据集

我们的解答步骤与上文相同，我们新增几个函数:

将当前样本集分割成特征 i 取值为 value 的一部分和取值不为 value 的一部分（二分）。

```
def split_dataset_2(dataset,axis,value):
    sub_dataset_1=[]
    sub_dataset_2=[]
    for feature_vector in dataset:
        if feature_vector[axis]==value:
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            sub_dataset_1.append(reduced_feature_vector)
        else:
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            sub_dataset_2.append(reduced_feature_vector)
    return sub_dataset_1,sub_dataset_2
```

选取最优划分特征

我们选择的最优划分特征为:

```
def choose_best_feature_to_split(dataset):
    num_features=len(dataset[0])-1 #特征总数
```

```
    if num_features==1: #当只有一个特征时
        return 0
    #初始化最佳基尼系数
    best_gini_index=1
    # 初始化最优特征
    best_feature=-1
    #遍历所有特征，寻找最优特征和该特征下的最优切分点
    for i in range(num_features):
        feature_list=[example[i] for example in dataset]
        unique_vals=set(feature_list) #去重，每个属性值唯一
        new_gini_index=0
        # Gini字典中的每个值代表以该值对应的键作为切分点对当前集合进行划分
                                                后的Gini系数
        for value in unique_vals: # 对于当前特征的每个取值
            # 先求由该值进行划分得到的两个子集
            sub_dataset_1,sub_dataset_2=split_dataset_2(dataset,i,value)
            # 求两个子集占原集合的比例系数prob1 prob2
            prob_1=len(sub_dataset_1)/float(len(dataset))
            prob_2 =len(sub_dataset_2) / float(len(dataset))
            gini_index_1=cal_gini_values(sub_dataset_1)# 计算子集1的Gini系
                                                    数
            gini_index_2=cal_gini_values(sub_dataset_2)# 计算子集2的Gini系
                                                    数
            # 计算由当前最优切分点划分后的最终Gini系数
            new_gini_index=prob_1*gini_index_1+prob_2*gini_index_2
            # 更新最优特征和最优切分点
            if new_gini_index<best_gini_index:
                best_gini_index=new_gini_index
                best_feature=i
                best_split_point=value
    return best_feature,best_split_point
```

我们得到最优划分特征为:

```
3 清晰
```

创建决策树

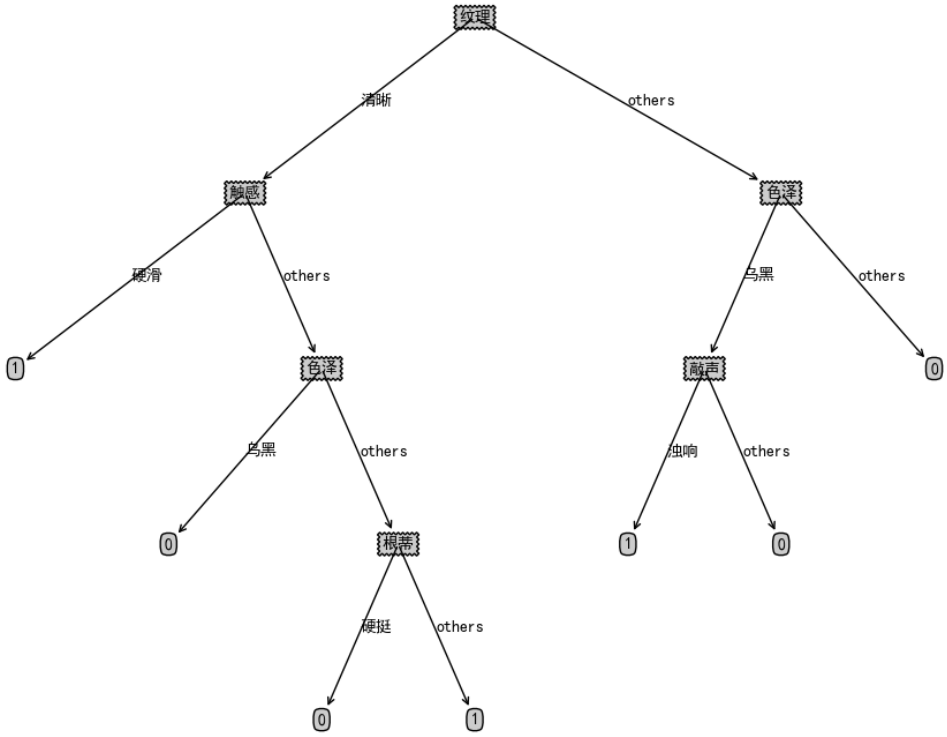
我们创建决策树如下:

```
def create_tree(dataset,features):
    class_list=[example[-1] for example in dataset]# 求出训练集所有样本的标
                                                   签
    # 先写两个递归结束的情况:
    # 若当前集合的所有样本标签相等（即样本已被分“纯”）
    # 则直接返回该标签值作为一个叶子节点
    if class_list.count(class_list[0])==len(class_list):
        return class_list[0]
    # 若训练集的所有特征都被使用完毕，当前无可用特征，但样本仍未被分“纯”
    # 则返回所含样本最多的标签作为结果
    if len(dataset[0])==1:
        return majority_cnt(class_list)
    # 下面是正式建树的过程
    # 选取进行分支的最佳特征的下标和最佳切分点
    best_feature,best_split_point=choose_best_feature_to_split(dataset)
    best_feature_label=features[best_feature]# 得到最佳特征
    my_tree={best_feature_label:{}}# 初始化决策树
    del(features[best_feature])# 使用过当前最佳特征后将其删去
    sub_labels=features[:]# 子特征 = 当前特征（因为刚才已经删去了用过的特
                                                   征）
    # 递归调用create_tree去生成新节点
    # 生成由最优切分点划分出来的二分子集
    sub_dataset_1,sub_dataset_2=split_dataset_2(dataset,best_feature,
                                                best_split_point)
    # 构造左子树
    my_tree[best_feature_label][best_split_point]=create_tree(sub_dataset_1
                                                ,sub_labels)
    # 构造右子树
    my_tree[best_feature_label]['others']=create_tree(sub_dataset_2,
                                                sub_labels)
    return my_tree
```

我们得到的决策树为:

```
{'纹理': {'清晰': {'触感': {'硬滑': '1', 'others': {'色泽': {'青绿': {'根蒂
                                    ': {'稍蜷': '1', 'others': '0'}}, '
                                    others': '0'}}}}, 'others': {'色泽':
                                    {'乌黑': {'敲声': {'沉闷': '0', '
                                    others': '1'}}, 'others': '0'}}}}
```

可视化决策树如下:



从上图我们看出，此时生成的决策树为二叉树。

决策树分类

```
def classify(decision_tree,features,test_example):
    # 根节点代表的属性
    first_feature = list(decision_tree.keys())[0]
    # second_dict是第一个分类属性的值（也是字典）
    second_dict = decision_tree[first_feature]
    # 树根代表的属性，所在属性标签中的位置，即第几个属性
    index_of_first_feature = features.index(first_feature)
    # 对于second_dict中的每一个key
    for key in second_dict.keys():
        # 不等于'others'的key
        if key != 'others':
            if test_example[index_of_first_feature] == key:
            # 若当前second_dict的key的value是一个字典
                if type(second_dict[key]).__name__ == 'dict':
```

```
                    # 则需要递归查询
                    class_label = classify(second_dict[key], features,
                                                        test_example)
                    # 若当前second_dict的key的value是一个单独的值
                else:
                    # 则就是要找的标签值
                    class_label = second_dict[key]
                # 如果测试样本在当前特征的取值不等于key，就说明它在当前特征
                                                        的取值属于'others'
            else:
                # 如果second_dict['others']的值是个字符串，则直接输出
                if isinstance(second_dict['others'], str):
                    class_label = second_dict['others']
                    # 如果second_dict['others']的值是个字典，则递归查询
                else:
                    class_label = classify(second_dict['others'],features,
                                                        test_example)
    return class_label
```

我们拿几组数据来测试一下我们的分类函数:

```
labels=['色泽', '根蒂', '敲声', '纹理', '脐部', '触感']
test_example_1=['青绿', '蜷缩', '浊响', '清晰', '凹陷', '硬滑']
test_example_2=['乌黑', '稍蜷', '浊响', '清晰', '稍凹', '软粘']
result_1=classify(my_tree,labels,test_example_1)
result_2=classify(my_tree,labels,test_example_2)
print('分类结果为'+'好瓜'if result_1=='1' else '分类结果为'+'坏瓜')
print('分类结果为'+'好瓜'if result_2=='1' else '分类结果为'+'坏瓜')
```

结果展示为:

```
分类结果为好瓜
分类结果为坏瓜
```

第二章　决策树剪枝

剪枝是决策树学习算法对于“过拟合”的主要手段。在决策树学习中，为了尽可能正确分类训练样本，节点划分过程将不断重复，有时会造成决策树分支过多，这时就可能因为训练样本学得“太好”了，以致于把训练集自身的一些特点当作所有的数据都具有的一般性质而导致过拟合。

因此，可通过自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的提升，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，然后自底向上的对非叶结点进行考察，若将该结点对应的子树替换成叶结点能带来决策树泛化性能提升，则将该子树替换成叶结点。

2.1　预剪枝决策树

预剪枝是在决策树生成过程中，在划分节点时，若该节点的划分没有提高其在训练集上的准确率，则不进行划分。

计算数据集的基尼指数

```
def cal_gini(dataset):
    num_examples = len(dataset)
    label_counts = {}
    # 给所有可能分类创建字典
    for feature_vector in dataset:
        current_label = feature_vector[-1]
        if current_label not in label_counts.keys():
            label_counts[current_label] = 0
```

```
        label_counts[current_label] += 1
    gini = 1.0
    for key in label_counts:
        prob = float(label_counts[key]) / num_examples
        gini -= prob * prob
    return gini
```

划分数据集

```
def split_dataset(dataset, axis, value):
    reduced_dataset = []
    for feature_vector in dataset:
        if feature_vector[axis] == value:
            reduced_feature_vector = feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis + 1:])
            reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset


# 对连续变量划分数据集, direction规定划分的方向,
# 决定是划分出小于value的数据样本还是大于value的数据样本集
def split_continuous_dataset(dataset, axis, value, direction):
    reduced_dataset = []
    for feature_vector in dataset:
        if direction == 0:
            if feature_vector[axis] > value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
        else:
            if feature_vector[axis] <= value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset
```

选择最好的数据集划分方式

```
def choose_best_feature_to_split(dataset, labels):
    num_features = len(dataset[0]) - 1
    best_gini_index = 1
    best_feature = -1
    best_split_dict = {}
    for i in range(num_features):
        feature_list = [example[i] for example in dataset]
        # 对连续型特征进行处理
        if type(feature_list[0]).__name__ == 'float' or type(feature_list[0
                                            ]).__name__ == 'int':
            # 产生n-1个候选划分点
            sorted_feature_list = sorted(feature_list)
            split_list = []
            for j in range(len(sorted_feature_list) - 1):
                split_list.append((sorted_feature_list[j] +
                                                    sorted_feature_list[j
                                                     + 1]) / 2.0)
            best_split_gini = 10000
            split_len = len(split_list)
            # 求用第j个候选划分点划分时，得到的信息熵，并记录最佳划分点
            for j in range(split_len):
                value = split_list[j]
                new_gini_index = 0.0
                sub_dataset_0 = split_continuous_dataset(dataset, i, value,
                                                         0)
                sub_dataset_1 = split_continuous_dataset(dataset, i, value,
                                                         1)
                prob0 = len(sub_dataset_0) / float(len(dataset))
                new_gini_index += prob0 * cal_gini(sub_dataset_0)
                prob1 = len(sub_dataset_1) / float(len(dataset))
                new_gini_index += prob1 * cal_gini(sub_dataset_1)
                if new_gini_index < best_split_gini:
                    best_split_gini = new_gini_index
                    best_split = j
            # 用字典记录当前特征的最佳划分点
            best_split_dict[labels[i]] = split_list[best_split]
            gini_index = best_split_gini
```

```
        # 对离散型特征进行处理
        else:
            unique_values = set(feature_list)
            new_gini_index = 0.0
            # 计算该特征下每种划分的信息熵
            for value in unique_values:
                sub_dataset = split_dataset(dataset, i, value)
                prob = len(sub_dataset) / float(len(dataset))
                new_gini_index += prob * cal_gini(sub_dataset)
            gini_index = new_gini_index
        if gini_index < best_gini_index:
            best_gini_index = gini_index
            best_feature = i
    # 若当前节点的最佳划分特征为连续特征，则将其以之前记录的划分点为界进行
                                            二值化处理
    # 即是否小于等于bestSplitValue
    # 并将特征名改为 name<=value的格式
    if type(dataset[0][best_feature]).__name__ == 'float' or type(dataset[0
                                        ][best_feature]).__name__ == 'int
                                        ':
        best_split_value = best_split_dict[labels[best_feature]]
        labels[best_feature] = labels[best_feature] + '<=' + str(
                                            best_split_value)
        for i in range(shape(dataset)[0]):
            if dataset[i][best_feature] <= best_split_value:
                dataset[i][best_feature] = 1
            else:
                dataset[i][best_feature] = 0
    return best_feature
```

节点下的样本投票

```
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1
```

```
    sorted_class_count = sorted(class_count.items(), key=operator.
                                        itemgetter(1), reverse=True)
    return sorted_class_count[0][0]
```

决策树分类函数

```
# 由于在Tree中，连续值特征的名称以及改为了feature<=value的形式
# 因此对于这类特征，需要利用正则表达式进行分割，获得特征名以及分割阈值
def classify(input_tree, feature_labels, test_vector):
    first_feature = list(input_tree.keys())[0]
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                            group()[2:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                            2]
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(feature_key)
        if test_vector[feature_index] <= feature_value:
            judge = 1
        else:
            judge = 0
        for key in second_dict.keys():
            if judge == int(key):
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                    , test_vector)
                else:
                    class_label = second_dict[key]
    else:
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(first_feature)
        for key in second_dict.keys():
            if test_vector[feature_index] == key:
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                    , test_vector)
                else:
                    class_label = second_dict[key]
    return class_label
```

测试决策树正确率

```
def testing(my_tree, data_test, labels):
    error = 0.0
    for i in range(len(data_test)):
        if classify(my_tree, labels, data_test[i]) != data_test[i][-1]:
            error += 1
    return float(error)
```

测试投票节点正确率

```
def testing_major(major, data_test):
    error = 0.0
    for i in range(len(data_test)):
        if major != data_test[i][-1]:
            error += 1
    return float(error)
```

递归产生决策树

```
def create_tree(dataset, labels, data_full, labels_full, data_test):
    class_list = [example[-1] for example in dataset]
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0]
    if len(dataset[0]) == 1:
        return majority_cnt(class_list)
    temp_labels = copy.deepcopy(labels)
    best_feature = choose_best_feature_to_split(dataset, labels)
    best_feature_label = labels[best_feature]
    my_tree = {best_feature_label: {}}
    if type(dataset[0][best_feature]).__name__ == 'str':
        current_label = labels_full.index(labels[best_feature])
        feature_values_full = [example[current_label] for example in
                                                data_full]
        unique_values_full = set(feature_values_full)
    feature_values = [example[best_feature] for example in dataset]
```

```
    unique_values = set(feature_values)
    del (labels[best_feature])
    # 针对bestFeat的每个取值，划分出一个子树。
    for value in unique_values:
        sub_labels = labels[:]
        if type(dataset[0][best_feature]).__name__ == 'str':
            unique_values_full.remove(value)
        my_tree[best_feature_label][value] = create_tree(split_dataset(
                                             dataset, best_feature, value)
                                             , sub_labels, data_full,
                                             labels_full,split_dataset(
                                             data_test, best_feature,
                                             value))
    if type(dataset[0][best_feature]).__name__ == 'str':
        for value in unique_values_full:
            my_tree[best_feature_label][value] = majority_cnt(class_list)
    # 进行测试，若划分没有提高准确率，则不进行划分，返回该节点的投票值作为
                                             节点类别
    if testing(my_tree, data_test, temp_labels) < testing_major(
                                            majority_cnt(class_list),
                                            data_test):
        return my_tree
    return majority_cnt(class_list)
```

数据集的构建

```
def create_data():
    with open('./2022-06-19data/data_word.csv', encoding='utf-8', newline='
                                            \n') as f:
        reader = csv.reader(f)  # 此处读取到的数据是将每行数据当做列表返回
                                             的
        data_1 = []
        for row in reader:  # 此时输出的是一行行的列表
            data_1.append(row)
    feature_labels=data_1[0][:-1]
    feature_labels_full=feature_labels[:]
    all_data=data_1[1:]
    all_data_full=all_data[:]
```

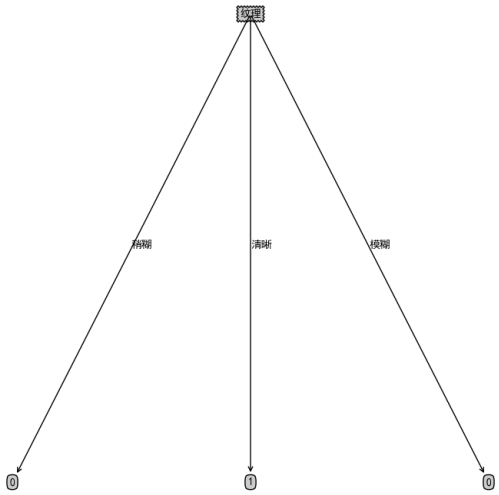
```
    with open('./2022-06-19data/train_data.csv', encoding='utf-8', newline=
                                            '\n') as f1:
        reader=csv.reader(f1)
        data_2=[]
        for row in reader:
            data_2.append(row)
    train_data=data_2[1:]
    train_data_full=train_data[:]
    with open('./2022-06-19data/test_data.csv',encoding='utf-8',newline='\n
                                            ') as f2:
        reader=csv.reader(f2)
        data_3=[]
        for row in reader:
            data_3.append(row)
    test_data=data_3[1:]
    return all_data,all_data_full,feature_labels,feature_labels_full,
                                            train_data,train_data_full,
                                            test_data
```

结果展示

我们预剪枝生成的决策树为:

```
{'纹理': {'稍糊': '0', '模糊': '0', '清晰': '1'}}
```

我们生成的决策树可视化为:



对比上图我们可以看出，预剪枝使得决策树的很多分支都没有展开，这不仅降低了过拟合的风险，还显著减少了决策树的训练时间开销和测试时间开销。

但是预剪枝决策树带来了欠拟合的风险。

2.2　后剪枝决策树

后剪枝决策树先生成一棵完整的决策树，再从底往顶进行剪枝处理。

决策树生成部分与未剪枝 CART 决策树相同，在其代码最后附上以下后剪枝代码即可：

分类函数

```
def classify(input_tree, feature_labels, test_vector):
    first_feature = list(input_tree.keys())[0]
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                                group()[1:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                                1]
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(feature_key)
        if test_vector[feature_index] <= feature_value:
            judge = 1
        else:
            judge = 0
        for key in second_dict.keys():
            if judge == int(key):
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                            , test_vector)
                else:
                    class_label = second_dict[key]
    else:
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(first_feature)
        for key in second_dict.keys():
            if test_vector[feature_index] == key:
                if type(second_dict[key]).__name__ == 'dict':
```

```
                    class_label = classify(second_dict[key], feature_labels
                                                            , test_vector)
                else:
                    class_label = second_dict[key]
    return class_label
```

测试决策树正确率和测试投票节点正确率与上面代码一样，这里不再赘述。

后剪枝

```
def post_pruning_tree(input_tree, dataset, data_test,feature_labels):
    first_feature = ''
    for i in input_tree.keys():
        first_feature=i
        break
    second_dict = input_tree[first_feature] #获取下一分支
    class_list = [example[-1] for example in dataset]
    feature_key = copy.deepcopy(first_feature)  #此特征值,和first_feature一
                                                样
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                                group()[2:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                                2]
    feature_index = feature_labels.index(feature_key)
    temp_labels = copy.deepcopy(feature_labels)
    del (feature_labels[feature_index])
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
                input_tree[first_feature][key] = post_pruning_tree(
                                                        second_dict[key],
                                                        split_dataset(dataset
                                                        , feature_index, key)
                                                        ,split_dataset(
                                                        data_test,
                                                        feature_index, key),
                                                        copy.deepcopy(
                                                        feature_labels))
    if testing(input_tree, data_test, temp_labels) <= testing_major(
```

```
                                        majority_cnt(class_list),
                                        data_test):
            return input_tree
        return majority_cnt(class_list)
```

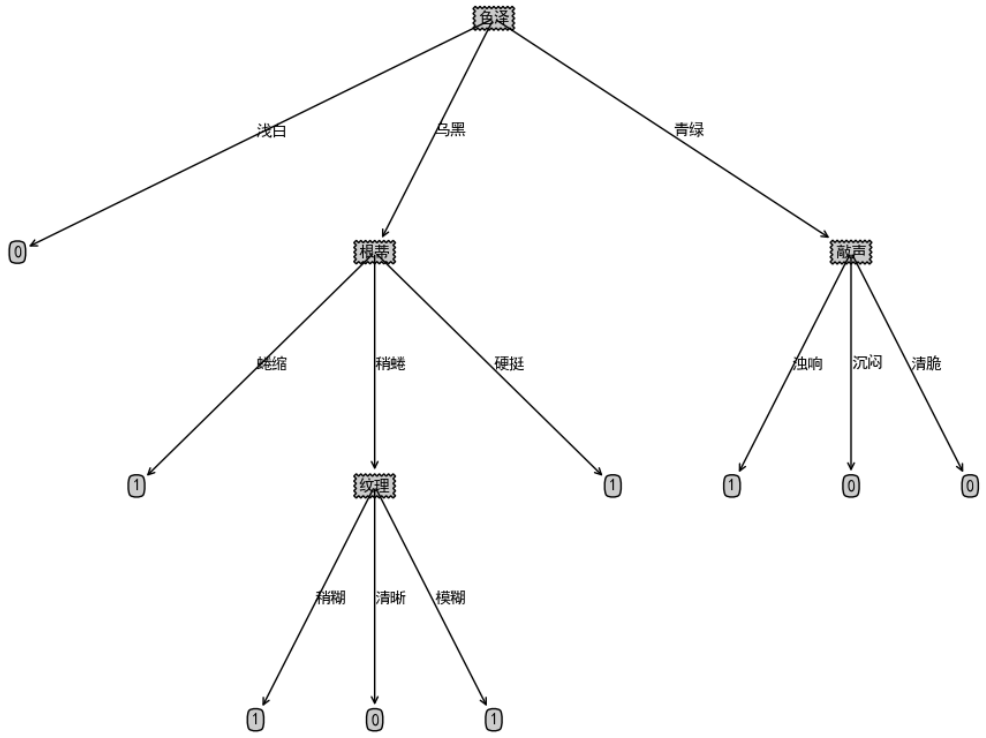
这里不同的是，我们把数据集划分，选择 10 个为训练集，另外七个为测试集。

```
def create_data():
    with open('./2022-06-19data/data_word.csv', encoding='utf-8', newline='
                                        \n') as f:
        reader = csv.reader(f)  # 此处读取到的数据是将每行数据当做列表返回
                                            的
        data_1 = []
        for row in reader:  # 此时输出的是一行行的列表
            data_1.append(row)
    feature_labels=data_1[0][:-1]
    feature_labels_full=feature_labels[:]
    all_data=data_1[1:]
    all_data_full=all_data[:]
    with open('./2022-06-19data/train_data.csv', encoding='utf-8', newline=
                                        '\n') as f1:
        reader=csv.reader(f1)
        data_2=[]
        for row in reader:
            data_2.append(row)
    train_data=data_2[1:]
    train_data_full=train_data[:]
    with open('./2022-06-19data/test_data.csv',encoding='utf-8',newline='\n
                                        ') as f2:
        reader=csv.reader(f2)
        data_3=[]
        for row in reader:
            data_3.append(row)
    test_data=data_3[1:]
    return all_data,all_data_full,feature_labels,feature_labels_full,
                                        train_data,train_data_full,
                                        test_data
```

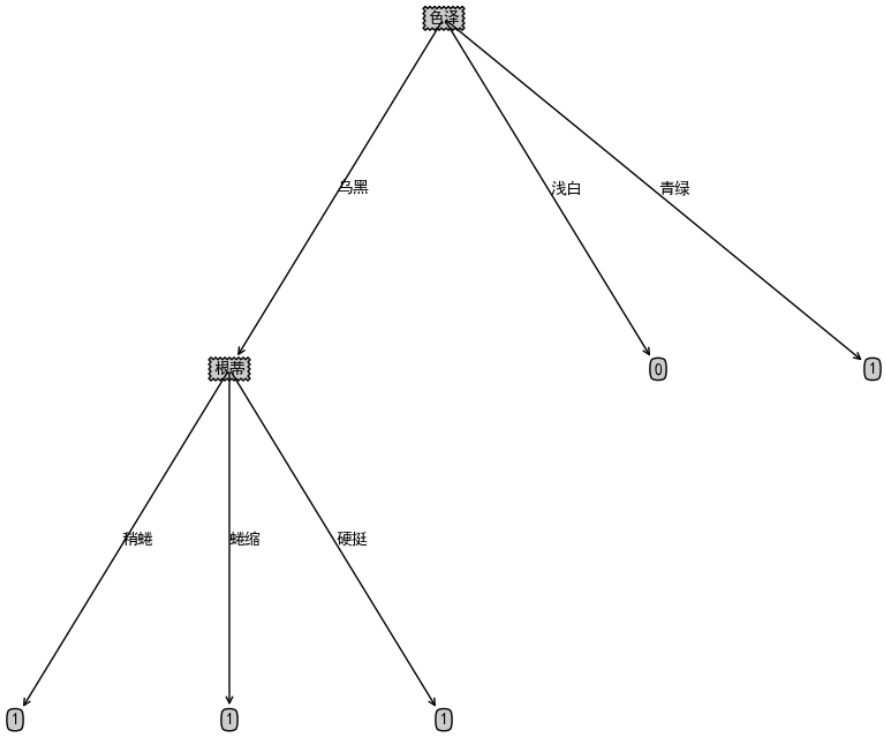
我们生成的决策树为:

```
myTree = create_tree(train_data, feature_labels, train_data_full,
                                        feature_labels_full)
print(myTree)
print(create_plot(myTree))
myTree2=post_pruning_tree(myTree,train_data,test_data,feature_labels_full)
print(myTree==myTree2)
print(create_plot(myTree2))
```

我们以训练集和测试集生成的决策树为:



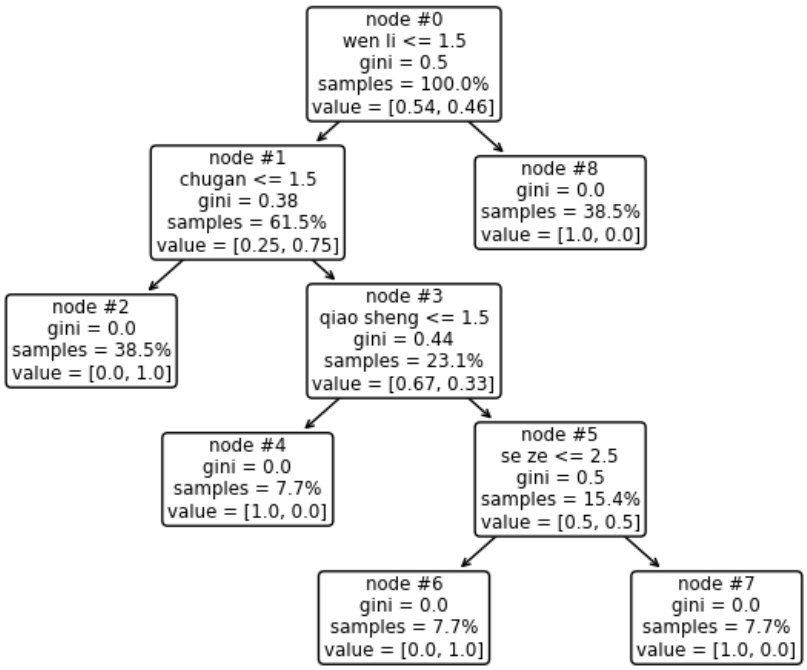
我们的后剪枝决策树为:



第三章　sklearn 实现决策树

3.1　普通分类树

```
df = pd.read_csv('./2022-06-19data/data_word.csv')
print(df)
#将特征值化为数字
df['色泽']=df['色泽'].map({'浅白':1,'青绿':2,'乌黑':3})
df['根蒂']=df['根蒂'].map({'稍蜷':1,'蜷缩':2,'硬挺':3})
df['敲声']=df['敲声'].map({'清脆':1,'浊响':2,'沉闷':3})
df['纹理']=df['纹理'].map({'清晰':1,'稍糊':2,'模糊':3})
df['脐部']=df['脐部'].map({'平坦':1,'稍凹':2,'凹陷':3})
df['触感'] = df['触感'].map({'硬滑':1,'软粘':2})
x_train,x_test,y_train,y_test=train_test_split(df[['色泽','根蒂','敲声','纹
                                        理','脐部','触感']],df['好瓜'],
                                        test_size=0.2)
gini=tree.DecisionTreeClassifier(random_state=0,max_depth=5)
gini.fit(x_train,y_train)
pred = gini.predict(x_test)
print('模型的预测准确率为:\n',accuracy_score(y_test,pred))
labels = ['se ze', 'gen di', 'qiao sheng', 'wen li', 'qi bu', 'chugan']
plot_tree(gini, feature_names=labels, node_ids=True,proportion=True,
                                        rounded=True, precision=2)
plt.tight_layout()
plt.show()
```



其中，模型的准确率为:

```
模型的预测准确率为:
 0.75
```

3.2　考虑模型复杂度

理论上，如果如果决策树不断地分裂，将使每个叶节点最终只有一个观测值（或多个相同的观测值），此时训练误差为 0，导致过拟合，泛化预测能力下降。因此，需要选择合适的决策树规模，停止分裂。一个好的解决方法是，先让决策树尽情生长，得到最大的数 T_{max}，再进行' 修枝'（pruning），以得到一个' 子树'（subtree）T。

对于任意子树 $T \in T_{max}$，定义其”复杂性“（complexity）为子树 T 的终节点数目，即为 |T|。为避免过拟合，不希望决策树过于复杂，故惩罚其规模 |T|:

$$min_T R(T) + \lambda \times |T|$$

其中，R(T) 为原来的损失函数，比如 0-1 损失函数，即在训练样本中，如果预测正确，则损失为 0，若预测错误则损失为 1。λ 为调节参数，也称为成本复杂性参数（cost-complexity-parameter, 简记 ccp）。λ 控制对决策树规模 |T| 的惩罚力度，可通过交叉验证确定。这种修枝方法称为成本复杂性修枝（cost-complexity pruning），即在成本（损失函数 R(T)) 与复杂性（决策树规模 |T|）之间进行最优权衡。

损失函数中惩罚项的理解：我们的目标是模型既预测准确，又相对简单，因此加入惩罚项是为了平衡预测准确性模型复杂性之间的平衡，如果增加一个变量使得残差平方和下降较多，而复杂度上升较小，从而使得损失函数下降较多，那么增加该变量可以接受的；如果增加一个变量只使残差平方和下降较少，而模型复杂度上升较多，那么增加该变量是不可接受的。其中模型复杂度的系数 λ 是非常关键的，如果 λ 很大，那么即使一个变量就将导致模型复杂度一个较大的上升，从而抑制变量的增加，反之则相反。

我们的代码实现如下:

```
model=DecisionTreeClassifier(random_state=0)
path=model.cost_complexity_pruning_path(x_train,y_train)
plt.plot(path.ccp_alphas, path.impurities, marker='o', drawstyle='steps-
                                        post')
plt.xlabel('alpha (cost-complexity parameter)')
plt.ylabel('Total Leaf Impurites')
plt.title('Total Leaf Impurites vs alpha for Training Set')
print("模型的复杂度参数与不纯度: ", max(path.ccp_alphas), max(path.
                                        impurities))
```

结果展示为:

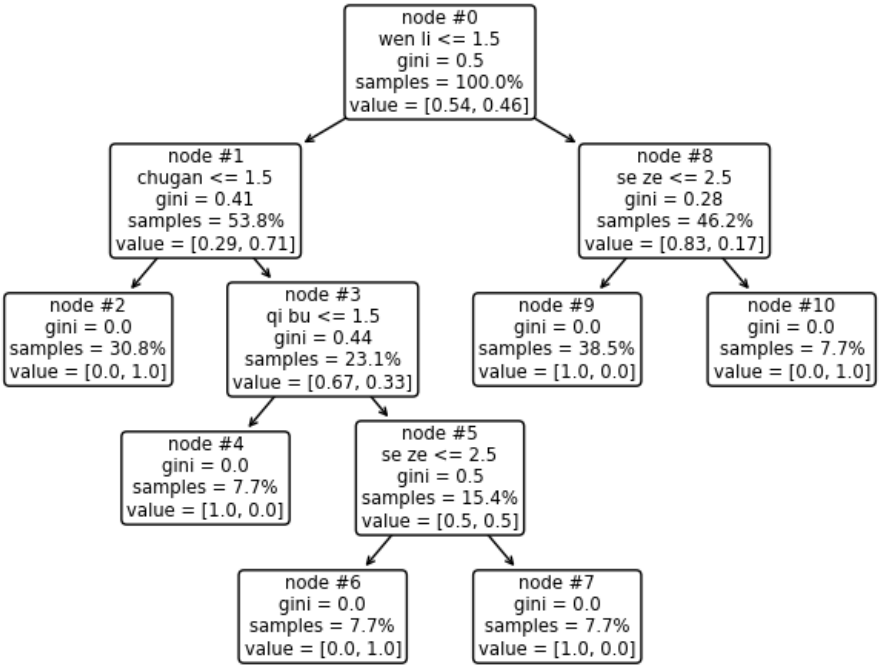
```
模型的复杂度参数与不纯度:  0.1844181459566075 0.4970414201183432
```

3.3 3 折交叉验证选择最优超参数

```
param_grid = {'ccp_alpha': path.ccp_alphas}
kfold = StratifiedKFold(n_splits=3,shuffle=True, random_state=1)
model_0 = GridSearchCV(DecisionTreeClassifier(random_state=123), param_grid
                                        , cv=kfold)
model_0.fit(x_train, y_train)
print("最优参数: ", model_0.best_params_)
model_1 = model_0.best_estimator_
print("预测准确率: ", model_1.score(x_test, y_test))
plot_tree(model_1, feature_names=labels, node_ids=True,proportion=True,
                                        rounded=True, precision=2)
plt.tight_layout()
plt.show()
```

我们的结果展示为:

```
最优参数：  {'ccp_alpha': 0.0}
预测准确率：  0.75
```



因为我们的数据集较少，因此没有剪枝的必要。

3.4　关于剪枝

Sklearn 中的决策树默认使用预剪枝操作来减小过拟合的风险，其中，预剪枝操作主要通过以下 5 个参数设置：

1 max-depth max-depth 限制树的最大深度，超过设定深度的树枝全部剪掉，一般用作树的”精修“这是用得最广泛的剪枝参数，在高维度低样本量时非常有效。决策树多生长一层，对样本量的需求会增加一倍，所以限制树深度能够有效地限制过拟合，在集成算法中也非常实用。常用的可以取值 10 左右。

2 min-samples-leaf 一个结点在分支后的每个子结点都必须包含至少 min-samples-leaf 个训练样本，否则分支就不会发生，或者，分支会朝着满足每个子结点都包含 min-samples-leaf 个样本的方向去发生。一般搭配 max-depth 使用，在回归树中有神奇的效果，可以让模型变得更加平滑。这个参数的数量设置得太小会引起过拟合，设置得太大就会阻止模型学习数据。一般来说，建议从 =5 开始使用。如果叶结点中含有的样本量

变化很大，建议输入浮点数作为样本量的百分比来使用。同时，这个参数可以保证每个叶子的最小尺寸，避免低方差，过拟合的叶子结点出现。

3 min-samples-split 一个结点必须要包含至少 min-samples-split 个训练样本，这个结点才允许被分支，否则分支就不会发生。

4 max-features max-features 限制分支时考虑的特征个数，超过限制个数的特征都会被舍弃。和 max-depth 异曲同工，max-features 是用来限制高维度数据的过拟合的剪枝参数，但其方法比较暴力，是直接限制可以使用的特征数量而强行使决策树停下的参数，在不知道决策树中的各个特征的重要性的情况下，强行设定这个参数可能会导致模型学习不足。如果希望通过降维的方式防止过拟合，建议使用 PCA，ICA 或者特征选择模块中的降维算法。

5 min-impurity-decrease min-impurity-decrease 限制信息增益的大小，信息增益小于设定数值的分支不会发生。这是在 0.19 版本中更新的功能，在 0.19 版本之前时使用 min-impurity-split。

第四章　源代码

4.1　多叉树

```
import csv
import operator
import numpy as np
import pandas as pd
from math import log
import matplotlib.pyplot as plt
import matplotlib as mpl

def create_data():
    """
    # dataSet中最后一列记录的是类别，其余列记录的是特征值
    dataSet = [
        ['青绿', '蜷缩', '浊响', '清晰', '凹陷', '硬滑', '1'],
        ['乌黑', '蜷缩', '沉闷', '清晰', '凹陷', '硬滑', '1'],
        ['乌黑', '蜷缩', '浊响', '清晰', '凹陷', '硬滑', '1'],
        ['青绿', '蜷缩', '沉闷', '清晰', '凹陷', '硬滑', '1'],
        ['浅白', '蜷缩', '浊响', '清晰', '凹陷', '硬滑', '1'],
        ['青绿', '稍蜷', '浊响', '清晰', '稍凹', '软粘', '1'],
        ['乌黑', '稍蜷', '浊响', '稍糊', '稍凹', '软粘', '1'],
        ['乌黑', '稍蜷', '浊响', '清晰', '稍凹', '硬滑', '1'],
        ['乌黑', '稍蜷', '沉闷', '稍糊', '稍凹', '硬滑', '0'],
        ['青绿', '硬挺', '清脆', '清晰', '平坦', '软粘', '0'],
        ['浅白', '硬挺', '清脆', '模糊', '平坦', '硬滑', '0'],
        ['浅白', '蜷缩', '浊响', '模糊', '平坦', '软粘', '0'],
        ['青绿', '稍蜷', '浊响', '稍糊', '凹陷', '硬滑', '0'],
        ['浅白', '稍蜷', '沉闷', '稍糊', '凹陷', '硬滑', '0'],
        ['乌黑', '稍蜷', '浊响', '清晰', '稍凹', '软粘', '0'],
        ['浅白', '蜷缩', '浊响', '模糊', '平坦', '硬滑', '0'],
```

```
        ['青绿', '蜷缩', '沉闷', '稍糊', '稍凹', '硬滑', '0']
    ]
    # label中记录的是特征的名称
    labels = ['色泽', '根蒂', '敲声', '纹理', '脐部', '触感']
    :return:
    """
    with open('./2022-06-19data/data_word.csv',encoding='utf-8',newline='\n
                                            ') as f:
        reader=csv.reader(f) #此处读取到的数据是将每行数据当做列表返回的
        data=[]
        for row in reader: #此时输出的是一行行的列表
            data.append(row)
    labels=data[0][:-1]
    dataset=data[1:]
    return dataset, labels

def CART_Gini(dataset):
    # 计算参与训练的数据量，即训练集中共有多少条数据;
    num_examples = len(dataset)
    # 计算每个分类标签label在当前节点出现的次数，即每个类别在当前节点下别分
                                            别出现多少次，用作信息熵概率 p 的
                                            分子
    label_counts = {}
    # 每次读入一条样本数据
    for feature_vector in dataset:
        # 将当前实例的类别存储，即每一行数据的最后一个数据代表的是类别
        current_label = feature_vector[-1]
        # 为所有分类创建字典，每个键值对都记录了当前类别出现的次数
        if current_label not in label_counts:
            label_counts[current_label] = 0
        label_counts[current_label] += 1
    # 使用循环方法，依次求出公式求和部分每个类别所占的比例及相应的计算
    gini = 1  # 记录基尼值
    for key in label_counts:
        # 算出该类别在该节点数据集中所占的比例
        prob = label_counts[key] / num_examples
        gini -= prob * prob
    return gini
```

```
'''
1.data为该节点的父节点使用的数据集
2.retDataSet为某特征下去除特定属性后其余的数据
3.value为某个特征下的其中一个属性
4.index为当前操作的是哪一个特征
'''
def split_data(dataset, axis, value):
    # 存储划分好的数据集
    reduced_dataset = []
    for feature_vector in dataset:
        if feature_vector[axis] == value:
            reduced_feature_vector = feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis + 1:])
            reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset

def choose_best_feature_to_split(dataset):
    # 求该数据集中共有多少特征（由于最后一列为label标签，所以减1）
    num_features = len(dataset[0]) - 1
    # 初始化最优基尼指数，和最优的特征索引
    best_gini_index, best_feature = 1, -1
    # 依次遍历每一个特征，计算其基尼指数
    # 当前数据集中有几列就说明有几个特征
    for i in range(num_features):
        # 将数据集中每一个特征下的所有属性拿出来生成一个列表
        feature_list = [example[i] for example in dataset]
        # 使用集合对列表进行去重，获得去重后的集合，即：此时集合中每个属性
                                                只出现一次
        unique_values = set(feature_list)
        # 创建一个临时基尼指数
        new_gini_index = 0
        # 依次遍历该特征下的所有属性
        for value in unique_values:
            # 依据每一个属性划分数据集
            sub_dataset = split_data(dataset, i, value)  # 详情见
                                                    splitDataSet函数
            # 计算该属性包含的数据占该特征包含数据的比例
            prob = len(sub_dataset) / len(dataset)
            # 计算每个属性结点的基尼值，并乘以相应的权重，再求他们的和，即
```

```
                                                    为该特征节点的基尼指数
            new_gini_index += prob * CART_Gini(sub_dataset)
            # 比较所有特征中的基尼指数，返回最好特征划分的索引值，注意：基
                                                    尼指数越小越好
        if (new_gini_index < best_gini_index):
            best_gini_index = new_gini_index
            best_feature = i
            # 返回最优的特征索引
    return best_feature

'''classList:为标签列的列表形式'''
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1  # classCount以字典形式记录每个类别出现的次
                                                    数
    # 倒叙排列classCount得到一个字典集合，然后取出第一个就是结果（好瓜/坏
                                                    瓜），即出现次数最多的结果
    # sortedClassCount的形式是[(),(),(),...],每个键值对变为元组并以列表形式
                                                    返回
    sorted_class_count = sorted(class_count.items(), key=operator.
                                                itemgetter(1), reverse=True)
    return sorted_class_count[0][0]  # 返回的是出现类别次数最多的“类别”

def create_tree(dataset, labels):
    # classList中记录了该节点数据中“类别”一列，保存为列表形式
    class_list = [example[-1] for example in dataset]
    # 如果该结点为空集，则将其设为叶子结点，节点类型为其父节点中类别最多的
                                                    类。
    if len(dataset) == 0:
        return majority_cnt(class_list)
    # 第一个停止条件：所有的类标签完全相同，则直接返回该类标签
    # 如果数据集到最后一列的第一个值出现的次数=整个集合的数量，也就说只有一
                                                    个类别，直接返回结果即可
    if class_list.count(class_list[0]) == len(class_list):  # count()函数是
                                                    统计括号中的值在list中出现的次数
        return class_list[0]
```

```
    # 第二个停止条件：使用完了所有特征，仍然不能将数据集划分成仅包含唯一类别的分组
    # 如果最后只剩一个特征，那么出现相同label多的一类，作为结果
    if len(dataset[0]) == 1:  # 所有的特征都用完了，只剩下最后的标签列了
        return majority_cnt(class_list)
    # 选择最优的特征
    best_feature = choose_best_feature_to_split(dataset)  # 返回的是最优特征的索引
    # 获取最优特征
    best_feature_label = labels[best_feature]
    # 初始化决策树
    my_tree = {best_feature_label: {}}
    # 将使用过的特征数据删除
    del (labels[best_feature])
    # 在数据集中去除最优特征列，然后用最优特征的分支继续生成决策树
    feature_values = [example[best_feature] for example in dataset]
    unique_values = set(feature_values)
    # 遍历该特征下每个属性节点，继续生成决策树
    for value in unique_values:
        # 求出剩余的可用的特征
        sub_labels = labels[:]
        my_tree[best_feature_label][value] = create_tree(split_data(dataset
                                                                    , best_feature, value),
                                                         sub_labels)
    return my_tree

#绘图相关参数的设置
def plot_node(node_txt, center_pt, parent_pt, node_type):
    # annotate函数是为绘制图上指定的数据点xy添加一个nodeTxt注释
    # nodeTxt是给数据点xy添加一个注释，xy为数据点的开始绘制的坐标,位于节点的中间位置
    # xycoords设置指定点xy的坐标类型，xytext为注释的中间点坐标，textcoords设置注释点坐标样式
    # bbox设置装注释盒子的样式,arrowprops设置箭头的样式
    '''
    figure points:表示坐标原点在图的左下角的数据点
    figure pixels:表示坐标原点在图的左下角的像素点
    figure fraction: 此时取值是小数，范围是([0,1],[0,1]),在图的左下角时xy是(0,0)，最右上角是(1,1)
```

```
    其他位置是按相对图的宽高的比例取最小值
    axes points : 表示坐标原点在图中坐标的左下角的数据点
    axes pixels : 表示坐标原点在图中坐标的左下角的像素点
    axes fraction : 与figure fraction类似，只不过相对于图的位置改成是相对于
                                            坐标轴的位置
    '''
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                            fraction',xytext=center_pt,
                                            textcoords='axes fraction',va="
                                            center", ha="center", bbox=
                                            node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化树的叶子节点个数
    # python3中my——tree.keys()返回的是dict_keys,不在是list,所以不能使用
                                            myTree.keys()[0]的方法获取结点属
                                            性，可以使用list(myTree.keys())[0
                                            ]
    first_str = list(my_tree.keys())[0]
    # 通过键名获取与之对应的值
    second_dict = my_tree[first_str]
    # 遍历树，secondDict.keys()获取所有的键
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                典，如果不是字典，代表此结点
                                                为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else: #如果不是字典，则叶子结点的数目就加1
            num_leafs += 1
    return num_leafs #返回叶子节点的数目
```

```
"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0] #获取树的第一个键名
    second_dict = my_tree[first_str] #获取下一个字典,获取键名所对应的值
    for key in second_dict.keys(): #遍历树
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1 #如果获取的键
                                                             是字典，树的深度加1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #去深度的最大值
            max_depth = thisDepth
    return max_depth #返回树的深度

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
#绘制线中间的文字(0和1)的绘制
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0] #计算文字的x坐标
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1] #计算文字的y坐标
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
```

```
    Parameters:
        my_tree - 决策树(字典)
        parent_pt - 标注的内容
        node_txt - 结点名
    Returns:
        无
    """
    #设置画节点用的盒子的样式
    leaf_node = dict(boxstyle="round4", fc="0.8")
    decision_node = dict(boxstyle="sawtooth", fc="0.8")

    #绘制树
    def plot_tree(my_tree, parent_pt, node_txt):
        num_leafs = get_num_leafs(my_tree) #获取树的叶子节点
        depth = get_tree_depth(my_tree) #获取树的深度
        first_str = list(my_tree.keys())[0] #获取第一个键名
        cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                            .total_w, plot_tree.y_off) #计算
                                            子节点的坐标
        plot_mid_text(cntr_pt, parent_pt, node_txt) #绘制线上的文字
        plot_node(first_str, cntr_pt, parent_pt, decision_node) #绘制节点
        second_dict = my_tree[first_str] #获取第一个键值
        plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d #计算节点y
                                            方向上的偏移量，根据树的深度
        for key in second_dict.keys():
            if type(second_dict[key]).__name__ == 'dict':
                plot_tree(second_dict[key], cntr_pt, str(key)) #递归绘制树
            else:
                # 更新x的偏移量,每个叶子结点x轴方向上的距离为 1/plotTree.totalW
                plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
                # 绘制非叶子节点
                plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                    cntr_pt, leaf_node)
                # 绘制箭头上的标志
                plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                                    key))
        plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

    """
```

```
    函数说明:创建绘制面板
    Parameters:
        in_tree - 决策树(字典)
    Returns:
        无
    """
    mpl.rcParams['font.family'] = 'SimHei'
    plt.rcParams['axes.unicode_minus'] = False

    def create_plot(in_tree):
        fig = plt.figure(1,figsize=(12,8),facecolor='white') #新建一个figure设
                                                置背景颜色为白色
        fig.clf() #清除figure
        axprops = dict(xticks=[], yticks=[])
        # 创建一个1行1列1个figure, 并把网格里面的第一个figure的Axes实例返回给
                                                ax1作为函数createPlot()
        # 的属性, 这个属性ax1相当于一个全局变量, 可以给plotNode函数使用
        create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
        plot_tree.total_w = float(get_num_leafs(in_tree)) #获取树的叶子节点
        plot_tree.total_d = float(get_tree_depth(in_tree)) #获取树的深度
        #节点的x轴的偏移量为-1/plotTree.totlaW/2,1为x轴的长度, 除以2保证每一个
                                                节点的x轴之间的距离为1/plotTree.
                                                totlaW*2
        plot_tree.x_off = -0.5 / plot_tree.total_w
        plot_tree.y_off = 1.0
        plot_tree(in_tree, (0.5, 1.0), '')
        plt.show()

    # 用上面训练好的决策树对新样本分类

    def classify(my_tree,test_data):
        first_feature=list(my_tree.keys())[0] #获取根节点
        second_dict=my_tree[first_feature] #获取下一级分支
        #查找当前列表中第一个匹配first_feature变量的元素的索引
        input_first=test_data.get(first_feature)
        input_value=second_dict[input_first] #获取测试样本通过第一个特征分类器
                                                后的输出
        if isinstance(input_value,dict): #判断结点是否为字典来判断是否为根节点
            class_label=classify(input_value,test_data)
```

```
    else:
        class_label=input_value #如果到达叶子结点，则返回当前节点的分类标签
    return class_label

dataset, labels=create_data()
print('数据集为:\n',dataset)
print('数据标签为:\n',labels)
my_tree=create_tree(dataset,labels)
print(my_tree)
test_data_1 = {'色泽': '青绿', '根蒂': '蜷缩', '敲声': '浊响', '纹理': '清
                                        晰', '脐部': '平坦', '触感': '硬滑'}
result=classify(my_tree,test_data_1)
print('分类结果为'+'好瓜'if result=='1' else '分类结果为'+'坏瓜')
print(create_plot(my_tree))
```

4.2　二叉树

```
import csv
import operator
import numpy as np
import pandas as pd
from math import log
import matplotlib.pyplot as plt
import matplotlib as mpl
file_name='./2022-06-19data/data_word.csv'
data=pd.read_csv(file_name)

def create_data():
    with open('./2022-06-19data/data_word.csv',encoding='utf-8',newline='\n
                                            ') as f:
        reader=csv.reader(f) #此处读取到的数据是将每行数据当做列表返回的
        data=[]
        for row in reader: #此时输出的是一行行的列表
            data.append(row)
    features=data[0][:-1]
    dataset=data[1:]
    return dataset,features
```

```
dataset,features=create_data()
print('数据集为:\n',dataset)
print('=='*30)
print('数据标签为:\n',features,type(features))
print('=='*30)

def cal_gini_values(dataset):
    # 求总样本数
    num_examples=len(dataset)
    label_count={} #初始化一个字典用来保存每个标签出现的次数
    for feature_vector in dataset:
        current_label=feature_vector[-1] #逐个获取标签信息
        if current_label not in label_count.keys():
            #如果标签没有放入统计次数字典的话，就添加进去
            label_count[current_label]=0
        label_count[current_label]+=1
    gini=1.0
    for key in label_count:
        prob=float(label_count[key])/num_examples
        gini-=prob*prob
    return gini

gini=cal_gini_values(dataset)
print('基尼指数的值为:\n',gini)
print('=='*30)

# 提取子集合
# 功能: 从dataSet中先找到所有第axis个标签值 = value的样本
# 然后将这些样本删去第axis个标签值，再全部提取出来成为一个新的样本集
"""
三个输入参数为: 待划分的数据集、划分数据集的特征、需要返回的特征的值。
第4行，如果第axis个特征满足分类的条件，则进行以下操作:
第5行，featVec[:axis]是从0号元素开始取axis个元素，此时reducedFeatVec是前
                                        axis个元素，即0号到axis-1号元素;
第6行，featVec[axis+1:]是从axis+1号元素开始取直到最后一个。extend函数将两次
                                        取的元素拼接起来，即从原来的列表中去
                                        掉了axis号元素;
第7行，将去除元素后的列表再组合起来，成为一个新的列表，即满足第axis个特征的
                                        列表。
```

```
由此，完成了对第axis个特征的划分。
"""
def split_dataset(dataset,axis,value):
    sub_dataset=[]
    for feature_vector in dataset:
        if feature_vector[axis]==value:
            #下面两句将axis特征去掉，并将符合条件的添加到返回的数据集中
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            sub_dataset.append(reduced_feature_vector)
    return sub_dataset

"""
将当前样本集分割成特征i取值为value的一部分和取值不为value的一部分（二分）
"""
def split_dataset_2(dataset,axis,value):
    sub_dataset_1=[]
    sub_dataset_2=[]
    for feature_vector in dataset:
        if feature_vector[axis]==value:
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            sub_dataset_1.append(reduced_feature_vector)
        else:
            reduced_feature_vector=feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis+1:])
            sub_dataset_2.append(reduced_feature_vector)
    return sub_dataset_1,sub_dataset_2

def choose_best_feature_to_split(dataset):
    num_features=len(dataset[0])-1 #特征总数
    if num_features==1: #当只有一个特征时
        return 0
    #初始化最佳基尼系数
    best_gini_index=1
    # 初始化最优特征
    best_feature=-1
    #遍历所有特征，寻找最优特征和该特征下的最优切分点
    for i in range(num_features):
```

```
        feature_list=[example[i] for example in dataset]
        unique_vals=set(feature_list) #去重，每个属性值唯一
        new_gini_index=0
        # Gini字典中的每个值代表以该值对应的键作为切分点对当前集合进行划分
                                                后的Gini系数
        for value in unique_vals: # 对于当前特征的每个取值
            # 先求由该值进行划分得到的两个子集
            sub_dataset_1,sub_dataset_2=split_dataset_2(dataset,i,value)
            # 求两个子集占原集合的比例系数prob1 prob2
            prob_1=len(sub_dataset_1)/float(len(dataset))
            prob_2 =len(sub_dataset_2) / float(len(dataset))
            gini_index_1=cal_gini_values(sub_dataset_1)# 计算子集1的Gini系
                                                    数
            gini_index_2=cal_gini_values(sub_dataset_2)# 计算子集2的Gini系
                                                    数
            # 计算由当前最优切分点划分后的最终Gini系数
            new_gini_index=prob_1*gini_index_1+prob_2*gini_index_2
            # 更新最优特征和最优切分点
            if new_gini_index<best_gini_index:
                best_gini_index=new_gini_index
                best_feature=i
                best_split_point=value
    return best_feature,best_split_point

best_feature,best_split_point=choose_best_feature_to_split(dataset)
print(best_feature,best_split_point)
print('=='*30)

#特征若已经划分完，节点下的样本还没有统一取值，则需要进行投票
# 初始化统计各标签次数的字典
# 键为各标签，对应的值为标签出现的次数
def majority_cnt(class_list):
    class_count={}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote]=0
        class_count[vote]+=1
    # 将classCount按值降序排列
    sorted_class_count=sorted(class_count.items(),key=operator.itemgetter(1
```

```
                                        ),reverse=True)
    return sorted_class_count[0][0] # 取sorted_labelCnt中第一个元素中的第一
                                        个值，即为所求


def create_tree(dataset,features):
    class_list=[example[-1] for example in dataset]# 求出训练集所有样本的标
                                        签
    # 先写两个递归结束的情况：
    # 若当前集合的所有样本标签相等（即样本已被分“纯”）
    # 则直接返回该标签值作为一个叶子节点
    if class_list.count(class_list[0])==len(class_list):
        return class_list[0]
    # 若训练集的所有特征都被使用完毕，当前无可用特征，但样本仍未被分“纯”
    # 则返回所含样本最多的标签作为结果
    if len(dataset[0])==1:
        return majority_cnt(class_list)
    # 下面是正式建树的过程
    # 选取进行分支的最佳特征的下标和最佳切分点
    best_feature,best_split_point=choose_best_feature_to_split(dataset)
    best_feature_label=features[best_feature]# 得到最佳特征
    my_tree={best_feature_label:{}}# 初始化决策树
    del(features[best_feature])# 使用过当前最佳特征后将其删去
    sub_labels=features[:]# 子特征 = 当前特征（因为刚才已经删去了用过的特
                                        征）
    # 递归调用create_tree去生成新节点
    # 生成由最优切分点划分出来的二分子集
    sub_dataset_1,sub_dataset_2=split_dataset_2(dataset,best_feature,
                                        best_split_point)
    # 构造左子树
    my_tree[best_feature_label][best_split_point]=create_tree(sub_dataset_1
                                        ,sub_labels)
    # 构造右子树
    my_tree[best_feature_label]['others']=create_tree(sub_dataset_2,
                                        sub_labels)
    return my_tree

my_tree=create_tree(dataset,features)
print(my_tree)
print('=='*30)
```

```
def plot_node(node_txt, center_pt, parent_pt, node_type):
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                        fraction',xytext=center_pt,
                                        textcoords='axes fraction',va="
                                        center", ha="center", bbox=
                                        node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化叶子
    first_str = list(my_tree.keys())[0] ##python3中my——tree.keys()返回的
                                        是dict_keys,不在是list,所以不能使
                                        用myTree.keys()[0]的方法获取结点
                                        属性，可以使用list(myTree.keys())
                                        [0]
    second_dict = my_tree[first_str] ##获取下一组字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                典，如果不是字典，代表此结点
                                                为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs

"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
```

```
"""
def get_tree_depth(my_tree):
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0]
    second_dict = my_tree[first_str] #获取下一个字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #更新层数
            max_depth = thisDepth
    return max_depth

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""
```

```
leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")

def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree)
    depth = get_tree_depth(my_tree)
    first_str = list(my_tree.keys())[0]
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                        .total_w, plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = my_tree[first_str]
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                               key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""

mpl.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
```

```
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree))
    plot_tree.total_d = float(get_tree_depth(in_tree))
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()

def classify(decision_tree,features,test_example):
    # 根节点代表的属性
    first_feature = list(decision_tree.keys())[0]
    # second_dict是第一个分类属性的值（也是字典）
    second_dict = decision_tree[first_feature]
    # 树根代表的属性，所在属性标签中的位置，即第几个属性
    index_of_first_feature = features.index(first_feature)
    # 对于second_dict中的每一个key
    for key in second_dict.keys():
        # 不等于'others'的key
        if key != 'others':
            if test_example[index_of_first_feature] == key:
            # 若当前second_dict的key的value是一个字典
                if type(second_dict[key]).__name__ == 'dict':
                    # 则需要递归查询
                    class_label = classify(second_dict[key], features,
                                                        test_example)
                    # 若当前second_dict的key的value是一个单独的值
                else:
                    # 则就是要找的标签值
                    class_label = second_dict[key]
                # 如果测试样本在当前特征的取值不等于key，就说明它在当前特征
                                                    的取值属于'others'
            else:
                # 如果second_dict['others']的值是个字符串，则直接输出
                if isinstance(second_dict['others'], str):
                    class_label = second_dict['others']
                    # 如果second_dict['others']的值是个字典，则递归查询
                else:
                    class_label = classify(second_dict['others'],features,
                                                        test_example)
```

```
    return class_label

labels=['色泽', '根蒂', '敲声', '纹理', '脐部', '触感']
test_example_1=['青绿', '蜷缩', '浊响', '清晰', '凹陷', '硬滑']
test_example_2=['乌黑', '稍蜷', '浊响', '清晰', '稍凹', '软粘']
result_1=classify(my_tree,labels,test_example_1)
result_2=classify(my_tree,labels,test_example_2)
print('分类结果为'+'好瓜'if result_1=='1' else '分类结果为'+'坏瓜')
print('分类结果为'+'好瓜'if result_2=='1' else '分类结果为'+'坏瓜')
print(create_plot(my_tree))
```

4.3　预剪枝

```
import pandas as pd
from numpy import *
import copy
import re
import matplotlib.pyplot as plt
import matplotlib as mpl
import csv
import operator
# 计算数据集的基尼指数
def cal_gini(dataset):
    num_examples = len(dataset)
    label_counts = {}
    # 给所有可能分类创建字典
    for feature_vector in dataset:
        current_label = feature_vector[-1]
        if current_label not in label_counts.keys():
            label_counts[current_label] = 0
        label_counts[current_label] += 1
    gini = 1.0
    for key in label_counts:
        prob = float(label_counts[key]) / num_examples
        gini -= prob * prob
    return gini

# 对离散变量划分数据集，取出该特征取值为value的所有样本
```

```
def split_dataset(dataset, axis, value):
    reduced_dataset = []
    for feature_vector in dataset:
        if feature_vector[axis] == value:
            reduced_feature_vector = feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis + 1:])
            reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset

# 对连续变量划分数据集, direction规定划分的方向,
# 决定是划分出小于value的数据样本还是大于value的数据样本集
def split_continuous_dataset(dataset, axis, value, direction):
    reduced_dataset = []
    for feature_vector in dataset:
        if direction == 0:
            if feature_vector[axis] > value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
        else:
            if feature_vector[axis] <= value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset

# 选择最好的数据集划分方式
def choose_best_feature_to_split(dataset, labels):
    num_features = len(dataset[0]) - 1
    best_gini_index = 1
    best_feature = -1
    best_split_dict = {}
    for i in range(num_features):
        feature_list = [example[i] for example in dataset]
        # 对连续型特征进行处理
        if type(feature_list[0]).__name__ == 'float' or type(feature_list[0
                                                ]).__name__ == 'int':
            # 产生n-1个候选划分点
            sorted_feature_list = sorted(feature_list)
```

```
            split_list = []
            for j in range(len(sorted_feature_list) - 1):
                split_list.append((sorted_feature_list[j] +
                                                    sorted_feature_list[j
                                                    + 1]) / 2.0)
            best_split_gini = 10000
            split_len = len(split_list)
            # 求用第j个候选划分点划分时，得到的信息熵，并记录最佳划分点
            for j in range(split_len):
                value = split_list[j]
                new_gini_index = 0.0
                sub_dataset_0 = split_continuous_dataset(dataset, i, value,
                                                    0)
                sub_dataset_1 = split_continuous_dataset(dataset, i, value,
                                                    1)
                prob0 = len(sub_dataset_0) / float(len(dataset))
                new_gini_index += prob0 * cal_gini(sub_dataset_0)
                prob1 = len(sub_dataset_1) / float(len(dataset))
                new_gini_index += prob1 * cal_gini(sub_dataset_1)
                if new_gini_index < best_split_gini:
                    best_split_gini = new_gini_index
                    best_split = j
            # 用字典记录当前特征的最佳划分点
            best_split_dict[labels[i]] = split_list[best_split]
            gini_index = best_split_gini
        # 对离散型特征进行处理
        else:
            unique_values = set(feature_list)
            new_gini_index = 0.0
            # 计算该特征下每种划分的信息熵
            for value in unique_values:
                sub_dataset = split_dataset(dataset, i, value)
                prob = len(sub_dataset) / float(len(dataset))
                new_gini_index += prob * cal_gini(sub_dataset)
            gini_index = new_gini_index
        if gini_index < best_gini_index:
            best_gini_index = gini_index
            best_feature = i
    # 若当前节点的最佳划分特征为连续特征，则将其以之前记录的划分点为界进行
```

```
                                        二值化处理
    # 即是否小于等于bestSplitValue
    # 并将特征名改为 name<=value的格式
    if type(dataset[0][best_feature]).__name__ == 'float' or type(dataset[0
                                        ][best_feature]).__name__ == 'int
                                        ':
        best_split_value = best_split_dict[labels[best_feature]]
        labels[best_feature] = labels[best_feature] + '<=' + str(
                                            best_split_value)
        for i in range(shape(dataset)[0]):
            if dataset[i][best_feature] <= best_split_value:
                dataset[i][best_feature] = 1
            else:
                dataset[i][best_feature] = 0
    return best_feature

# 特征若已经划分完，节点下的样本还没有统一取值，则需要进行投票
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1
    sorted_class_count = sorted(class_count.items(), key=operator.
                                        itemgetter(1), reverse=True)
    return sorted_class_count[0][0]

# 由于在Tree中，连续值特征的名称以及改为了feature<=value的形式
# 因此对于这类特征，需要利用正则表达式进行分割，获得特征名以及分割阈值
def classify(input_tree, feature_labels, test_vector):
    first_feature = list(input_tree.keys())[0]
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                            group()[2:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                            2]
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(feature_key)
        if test_vector[feature_index] <= feature_value:
```

```
            judge = 1
        else:
            judge = 0
        for key in second_dict.keys():
            if judge == int(key):
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                            , test_vector)
                else:
                    class_label = second_dict[key]
    else:
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(first_feature)
        for key in second_dict.keys():
            if test_vector[feature_index] == key:
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                            , test_vector)
                else:
                    class_label = second_dict[key]
    return class_label

def testing(my_tree, data_test, labels):
    error = 0.0
    for i in range(len(data_test)):
        if classify(my_tree, labels, data_test[i]) != data_test[i][-1]:
            error += 1
    return float(error)

def testing_major(major, data_test):
    error = 0.0
    for i in range(len(data_test)):
        if major != data_test[i][-1]:
            error += 1
    return float(error)

# 主程序, 递归产生决策树
def create_tree(dataset, labels, data_full, labels_full, data_test):
    class_list = [example[-1] for example in dataset]
```

```
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0]
    if len(dataset[0]) == 1:
        return majority_cnt(class_list)
    temp_labels = copy.deepcopy(labels)
    best_feature = choose_best_feature_to_split(dataset, labels)
    best_feature_label = labels[best_feature]
    my_tree = {best_feature_label: {}}
    if type(dataset[0][best_feature]).__name__ == 'str':
        current_label = labels_full.index(labels[best_feature])
        feature_values_full = [example[current_label] for example in
                                                data_full]
        unique_values_full = set(feature_values_full)
    feature_values = [example[best_feature] for example in dataset]
    unique_values = set(feature_values)
    del (labels[best_feature])
    # 针对bestFeat的每个取值，划分出一个子树。
    for value in unique_values:
        sub_labels = labels[:]
        if type(dataset[0][best_feature]).__name__ == 'str':
            unique_values_full.remove(value)
        my_tree[best_feature_label][value] = create_tree(split_dataset(
                                                dataset, best_feature, value)
                                                , sub_labels, data_full,
                                                labels_full,split_dataset(
                                                data_test, best_feature,
                                                value))
    if type(dataset[0][best_feature]).__name__ == 'str':
        for value in unique_values_full:
            my_tree[best_feature_label][value] = majority_cnt(class_list)
    # 进行测试，若划分没有提高准确率，则不进行划分，返回该节点的投票值作为
                                            节点类别
    if testing(my_tree, data_test, temp_labels) < testing_major(
                                            majority_cnt(class_list),
                                            data_test):
        return my_tree
    return majority_cnt(class_list)

def plot_node(node_txt, center_pt, parent_pt, node_type):
```

```
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                        fraction',xytext=center_pt,
                                        textcoords='axes fraction',va="
                                        center", ha="center", bbox=
                                        node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化叶子
    first_str = list(my_tree.keys())[0] #python3中my——tree.keys()返回的是
                                        dict_keys,不在是list,所以不能使用
                                        myTree.keys()[0]的方法获取结点属
                                        性，可以使用list(myTree.keys())[0
                                        ]
    second_dict = my_tree[first_str] #获取下一组字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs


"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
```

```
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0]
    second_dict = my_tree[first_str] #获取下一个字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #更新层数
            max_depth = thisDepth
    return max_depth

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""

leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")
```

```
def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree)
    depth = get_tree_depth(my_tree)
    first_str = list(my_tree.keys())[0]
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                          .total_w, plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = my_tree[first_str]
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                  cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                                 key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""
mpl.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree))
    plot_tree.total_d = float(get_tree_depth(in_tree))
```

```
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()

def create_data():
    with open('./2022-06-19data/data_word.csv', encoding='utf-8', newline='
                                            \n') as f:
        reader = csv.reader(f)  # 此处读取到的数据是将每行数据当做列表返回
                                                的
        data_1 = []
        for row in reader:  # 此时输出的是一行行的列表
            data_1.append(row)
    feature_labels=data_1[0][:-1]
    feature_labels_full=feature_labels[:]
    all_data=data_1[1:]
    all_data_full=all_data[:]
    with open('./2022-06-19data/train_data.csv', encoding='utf-8', newline=
                                            '\n') as f1:
        reader=csv.reader(f1)
        data_2=[]
        for row in reader:
            data_2.append(row)
    train_data=data_2[1:]
    train_data_full=train_data[:]
    with open('./2022-06-19data/test_data.csv',encoding='utf-8',newline='\n
                                            ') as f2:
        reader=csv.reader(f2)
        data_3=[]
        for row in reader:
            data_3.append(row)
    test_data=data_3[1:]
    return all_data,all_data_full,feature_labels,feature_labels_full,
                                            train_data,train_data_full,
                                            test_data

all_data,all_data_full,feature_labels,feature_labels_full,train_data,
                                    train_data_full,test_data=create_data
                                    ()
```

```
my_tree = create_tree(all_data, feature_labels, all_data_full,
                                        feature_labels_full, test_data)
print(my_tree)
print(create_plot(my_tree))
```

4.4　后剪枝

```
from numpy import *
import numpy as np
import pandas as pd
from math import log
import operator
import matplotlib.pyplot as plt
import matplotlib as mpl
import re
import copy
import csv

# 计算数据集的基尼指数
def cal_gini(dataset):
    num_examples = len(dataset)
    label_counts = {}
    # 给所有可能分类创建字典
    for feature_vector in dataset:
        current_label = feature_vector[-1]
        if current_label not in label_counts.keys():
            label_counts[current_label] = 0
        label_counts[current_label] += 1
    gini = 1.0
    for key in label_counts:
        prob = float(label_counts[key]) / num_examples
        gini -= prob * prob
    return gini

# 对离散变量划分数据集，取出该特征取值为value的所有样本
def split_dataset(dataset, axis, value):
    reduced_datadet = []
    for feature_vector in dataset:
```

```
        if feature_vector[axis] == value:
            reduced_feature_vector = feature_vector[:axis]
            reduced_feature_vector.extend(feature_vector[axis + 1:])
            reduced_datadet.append(reduced_feature_vector)
    return reduced_datadet

# 对连续变量划分数据集, direction规定划分的方向,
# 决定是划分出小于value的数据样本还是大于value的数据样本集
def split_continuous_dataset(dataset, axis, value, direction):
    reduced_dataset = []
    for feature_vector in dataset:
        if direction == 0:
            if feature_vector[axis] > value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
        else:
            if feature_vector[axis] <= value:
                reduced_feature_vector = feature_vector[:axis]
                reduced_feature_vector.extend(feature_vector[axis + 1:])
                reduced_dataset.append(reduced_feature_vector)
    return reduced_dataset

# 选择最好的数据集划分方式
def choose_best_feature_to_split(dataset, labels):
    num_features = len(dataset[0]) - 1
    best_gini_index = 100000.0
    best_feature = -1
    best_split_dict = {}
    for i in range(num_features):
        feature_list = [example[i] for example in dataset]
        # 对连续型特征进行处理
        if type(feature_list[0]).__name__ == 'float' or type(feature_list[0
                                                ]).__name__ == 'int':
            # 产生n-1个候选划分点
            sorted_feature_list = sorted(feature_list)
            split_list = []
            for j in range(len(sorted_feature_list) - 1):
                split_list.append((sorted_feature_list[j] +
```

```
                                                    sorted_feature_list[j
                                                    + 1]) / 2.0)
            best_split_gini = 10000
            slen = len(split_list)
            # 求用第j个候选划分点划分时，得到的信息熵，并记录最佳划分点
            for j in range(slen):
                value = split_list[j]
                new_gini_index = 0.0
                sub_dataset_0 = split_continuous_dataset(dataset, i, value,
                                                         0)
                sub_dataset_1 = split_continuous_dataset(dataset, i, value,
                                                         1)
                prob0 = len(sub_dataset_0) / float(len(dataset))
                new_gini_index += prob0 * cal_gini(sub_dataset_0)
                prob1 = len(sub_dataset_1) / float(len(dataset))
                new_gini_index += prob1 * cal_gini(sub_dataset_1)
                if new_gini_index < best_split_gini:
                    best_split_gini = new_gini_index
                    best_split = j
            # 用字典记录当前特征的最佳划分点
            best_split_dict[labels[i]] = split_list[best_split]

            gini_index = best_split_gini
        # 对离散型特征进行处理
        else:
            unique_values = set(feature_list)
            new_gini_index= 0.0
            # 计算该特征下每种划分的信息熵
            for value in unique_values:
                sub_dataset = split_dataset(dataset, i, value)
                prob = len(sub_dataset) / float(len(dataset))
                new_gini_index += prob * cal_gini(sub_dataset)
            gini_index = new_gini_index
        if gini_index < best_gini_index:
            best_gini_index = gini_index
            best_feature = i
    # 若当前节点的最佳划分特征为连续特征，则将其以之前记录的划分点为界进行
                                        二值化处理
    # 即是否小于等于bestSplitValue
```

```
    if type(dataset[0][best_feature]).__name__ == 'float' or type(dataset[0
                                        ][best_feature]).__name__ == 'int
                                        ':
        best_split_value = best_split_dict[labels[best_feature]]
        labels[best_feature] = labels[best_feature] + '<=' + str(
                                            best_split_value)
        for i in range(shape(dataset)[0]):
            if dataset[i][best_feature] <= best_split_value:
                dataset[i][best_feature] = 1
            else:
                dataset[i][best_feature] = 0
    return best_feature

# 特征若已经划分完，节点下的样本还没有统一取值，则需要进行投票
def majority_cnt(class_list):
    class_count = {}
    for vote in class_list:
        if vote not in class_count.keys():
            class_count[vote] = 0
        class_count[vote] += 1
    sorted_class_count = sorted(class_count.items(), key=operator.
                                        itemgetter(1), reverse=True)
    return sorted_class_count[0][0]

# 主程序，递归产生决策树
def create_tree(dataset, labels, data_full, labels_full):
    class_list = [example[-1] for example in dataset]
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0]
    if len(dataset[0]) == 1:
        return majority_cnt(class_list)
    best_feature = choose_best_feature_to_split(dataset, labels)
    best_feature_label = labels[best_feature]
    myTree = {best_feature_label: {}}
    feature_values = [example[best_feature] for example in dataset]
    unique_values = set(feature_values)
    if type(dataset[0][best_feature]).__name__ == 'str':
        current_label = labels_full.index(labels[best_feature])
        feature_values_full = [example[current_label] for example in
```

```
                                                    data_full]
        unique_values_full = set(feature_values_full)
    del (labels[best_feature])
    # 针对bestFeat的每个取值，划分出一个子树。
    for value in unique_values:
        sub_labels = labels[:]
        if type(dataset[0][best_feature]).__name__ == 'str':
            unique_values_full.remove(value)
        myTree[best_feature_label][value] = create_tree(split_dataset \
                                                        (dataset,

    if type(dataset[0][best_feature]).__name__ == 'str':
        for value in unique_values_full:
            myTree[best_feature_label][value] = majority_cnt(class_list)
    return myTree

def classify(input_tree, feature_labels, test_vector):
    first_feature = list(input_tree.keys())[0]
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                                    group()[2:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                                    2]
        second_dict = input_tree[first_feature]
```

```
        feature_index = feature_labels.index(feature_key)
        if test_vector[feature_index] <= feature_value:
            judge = 1
        else:
            judge = 0
        for key in second_dict.keys():
            if judge == int(key):
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                          , test_vector)
                else:
                    class_label = second_dict[key]
    else:
        second_dict = input_tree[first_feature]
        feature_index = feature_labels.index(first_feature)
        for key in second_dict.keys():
            if test_vector[feature_index] == key:
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feature_labels
                                                          , test_vector)
                else:
                    class_label = second_dict[key]
    return class_label

# 测试决策树正确率
def testing(my_tree, data_test, labels):
    error = 0.0
    for i in range(len(data_test)):
        if classify(my_tree, labels, data_test[i][:-1]) != data_test[i][-1]
                                                        :
            error += 1
    return float(error)

# 测试投票节点正确率
def testing_major(major, data_test):
    error = 0.0
    for i in range(len(data_test)):
        if major != data_test[i][-1]:
            error += 1
```

```
    # print 'major %d' %error
    return float(error)

# 后剪枝
def post_pruning_tree(input_tree,dataset,data_test,feature_labels):
    first_feature = ''
    for i in input_tree.keys():
        first_feature=i
        break
    second_dict = input_tree[first_feature] #获取下一分支
    class_list = [example[-1] for example in dataset]
    feature_key = copy.deepcopy(first_feature)  #此特征值,和first_feature一
                                                样
    if '<=' in first_feature:
        feature_value = float(re.compile("(<=.+)").search(first_feature).
                                                    group()[1:])
        feature_key = re.compile("(.+<=)").search(first_feature).group()[:-
                                                    1]
    feature_index = feature_labels.index(feature_key)
    temp_labels = copy.deepcopy(feature_labels)
    del (feature_labels[feature_index])
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
                input_tree[first_feature][key] = post_pruning_tree(
                                                        second_dict[key],
                                                        split_dataset(dataset
                                                        , feature_index, key)
                                                        ,split_dataset(
                                                        data_test,
                                                        feature_index, key),
                                                        copy.deepcopy(
                                                        feature_labels))
    if testing(input_tree, data_test, temp_labels) <= testing_major(
                                            majority_cnt(class_list),
                                            data_test):
        return input_tree
    return majority_cnt(class_list)

def plot_node(node_txt, center_pt, parent_pt, node_type):
```

```
    arrow_args = dict(arrowstyle="<-") #定义箭头格式
    # 绘制结点
    create_plot.ax1.annotate(node_txt, xy=parent_pt, xycoords='axes
                                        fraction',xytext=center_pt,
                                        textcoords='axes fraction',va="
                                        center", ha="center", bbox=
                                        node_type, arrowprops=arrow_args)
"""
函数说明:获取决策树叶子结点的数目
Parameters:
    myTree - 决策树
Returns:
    numLeafs - 决策树的叶子结点的数目
"""
def get_num_leafs(my_tree):
    num_leafs = 0 #初始化叶子
    first_str = list(my_tree.keys())[0] #python3中my——tree.keys()返回的是
                                        dict_keys,不在是list,所以不能使用
                                        myTree.keys()[0]的方法获取结点属
                                        性，可以使用list(myTree.keys())[0
                                        ]
    second_dict = my_tree[first_str] #获取下一组字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs


"""
函数说明:获取决策树的层数
Parameters:
    myTree - 决策树
Returns:
    maxDepth - 决策树的层数
"""
def get_tree_depth(my_tree):
```

```
    max_depth = 0 #初始化决策树深度
    first_str = list(my_tree.keys())[0]
    second_dict = my_tree[first_str] #获取下一个字典
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict': #测试该结点是否为字
                                                      典，如果不是字典，代表此结点
                                                      为叶子结点
            thisDepth = get_tree_depth(second_dict[key]) + 1
        else:
            thisDepth = 1
        if thisDepth > max_depth: #更新层数
            max_depth = thisDepth
    return max_depth

"""
函数说明:标注有向边属性值
Parameters:
    cntr_pt、parent_pt - 用于计算标注位置
    txt_string - 标注的内容
Returns:
    无
"""
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string)

"""
函数说明:绘制决策树
Parameters:
    my_tree - 决策树(字典)
    parent_pt - 标注的内容
    node_txt - 结点名
Returns:
    无
"""

leaf_node = dict(boxstyle="round4", fc="0.8")
decision_node = dict(boxstyle="sawtooth", fc="0.8")
```

```
def plot_tree(my_tree, parent_pt, node_txt):
    num_leafs = get_num_leafs(my_tree)
    depth = get_tree_depth(my_tree)
    first_str = list(my_tree.keys())[0]
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree
                                          .total_w, plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = my_tree[first_str]
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off),
                                                  cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(
                                                 key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

"""
函数说明:创建绘制面板
Parameters:
    in_tree - 决策树(字典)
Returns:
    无
"""
mpl.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

def create_plot(in_tree):
    fig = plt.figure(1,figsize=(12,8),facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(in_tree))
    plot_tree.total_d = float(get_tree_depth(in_tree))
```

```
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(in_tree, (0.5, 1.0), '')
    plt.show()

def create_data():
    with open('./2022-06-19data/data_word.csv', encoding='utf-8', newline='
                                            \n') as f:
        reader = csv.reader(f)  # 此处读取到的数据是将每行数据当做列表返回
                                                的
        data_1 = []
        for row in reader:  # 此时输出的是一行行的列表
            data_1.append(row)
    feature_labels=data_1[0][:-1]
    feature_labels_full=feature_labels[:]
    all_data=data_1[1:]
    all_data_full=all_data[:]
    with open('./2022-06-19data/train_data.csv', encoding='utf-8', newline=
                                            '\n') as f1:
        reader=csv.reader(f1)
        data_2=[]
        for row in reader:
            data_2.append(row)
    train_data=data_2[1:]
    train_data_full=train_data[:]
    with open('./2022-06-19data/test_data.csv',encoding='utf-8',newline='\n
                                            ') as f2:
        reader=csv.reader(f2)
        data_3=[]
        for row in reader:
            data_3.append(row)
    test_data=data_3[1:]
    return all_data,all_data_full,feature_labels,feature_labels_full,
                                            train_data,train_data_full,
                                            test_data

all_data,all_data_full,feature_labels,feature_labels_full,train_data,
                                        train_data_full,test_data=create_data
                                        ()
```

```
myTree = create_tree(train_data, feature_labels, train_data_full,
                                        feature_labels_full)
print(myTree)
print(create_plot(myTree))
myTree2=post_pruning_tree(myTree,train_data,test_data,feature_labels_full)
print(create_plot(myTree2))
```

4.5　sklearn 决策树

```
# 导入包
import pandas as pd
from sklearn import tree
import numpy as np
from sklearn.model_selection import train_test_split
import graphviz
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier,plot_tree
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import GridSearchCV


df = pd.read_csv('./2022-06-19data/data_word.csv')
print(df)
#将特征值化为数字
df['色泽']=df['色泽'].map({'浅白':1,'青绿':2,'乌黑':3})
df['根蒂']=df['根蒂'].map({'稍蜷':1,'蜷缩':2,'硬挺':3})
df['敲声']=df['敲声'].map({'清脆':1,'浊响':2,'沉闷':3})
df['纹理']=df['纹理'].map({'清晰':1,'稍糊':2,'模糊':3})
df['脐部']=df['脐部'].map({'平坦':1,'稍凹':2,'凹陷':3})
df['触感'] = df['触感'].map({'硬滑':1,'软粘':2})
x_train,x_test,y_train,y_test=train_test_split(df[['色泽','根蒂','敲声','纹
                                        理','脐部','触感']],df['好瓜'],
                                        test_size=0.2)
gini=tree.DecisionTreeClassifier(random_state=0,max_depth=5)
gini.fit(x_train,y_train)
pred = gini.predict(x_test)
print('模型的预测准确率为:\n',accuracy_score(y_test,pred))
```

```
labels = ['se ze', 'gen di', 'qiao sheng', 'wen li', 'qi bu', 'chugan']
model=DecisionTreeClassifier(random_state=0)
path=model.cost_complexity_pruning_path(x_train,y_train)
plt.plot(path.ccp_alphas, path.impurities, marker='o', drawstyle='steps-
                                        post')
plt.xlabel('alpha (cost-complexity parameter)')
plt.ylabel('Total Leaf Impurites')
plt.title('Total Leaf Impurites vs alpha for Training Set')
print("模型的复杂度参数与不纯度: ", max(path.ccp_alphas), max(path.
                                        impurities))

param_grid = {'ccp_alpha': path.ccp_alphas}
kfold = StratifiedKFold(n_splits=3,shuffle=True, random_state=1)
model_0 = GridSearchCV(DecisionTreeClassifier(random_state=123), param_grid
                                        , cv=kfold)
model_0.fit(x_train, y_train)
print("最优参数: ", model_0.best_params_)
model_1 = model_0.best_estimator_
print("预测准确率: ", model_1.score(x_test, y_test))
plot_tree(model_1, feature_names=labels, node_ids=True,proportion=True,
                                        rounded=True, precision=2)
plt.tight_layout()
plt.show()
```