山东大学＿＿＿＿＿计算机科学与技术＿＿＿＿＿＿学院

＿机器学习与模式识别＿＿课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：Experiment 9: Decision Tree		
实验学时：2	实验日期：2025/4/8	

实验环境：

软件环境：

系统：Windows 11 家庭中文版 23H2 22631.4317

计算软件：MATLAB 版本: 9.8.0.1323502 (R2020a)

Java 版本: Java 1.8.0_202-b08 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode

硬件环境：

CPU：13th Gen Intel(R) Core(TM) i9-13980HX 2.20 GHz

内存：32.0 GB (31.6 GB 可用)

磁盘驱动器：NVMe WD_BLACKSN850X2000GB

显示适配器：NVIDIA GeForce RTX 4080 Laptop GPU

1. 实验内容

In this exercise, you need to implement Decision Tree.

2. 实验步骤

（1） 加载数据

（2） 数据预处理和测试集/训练集划分

（3） 实现算法

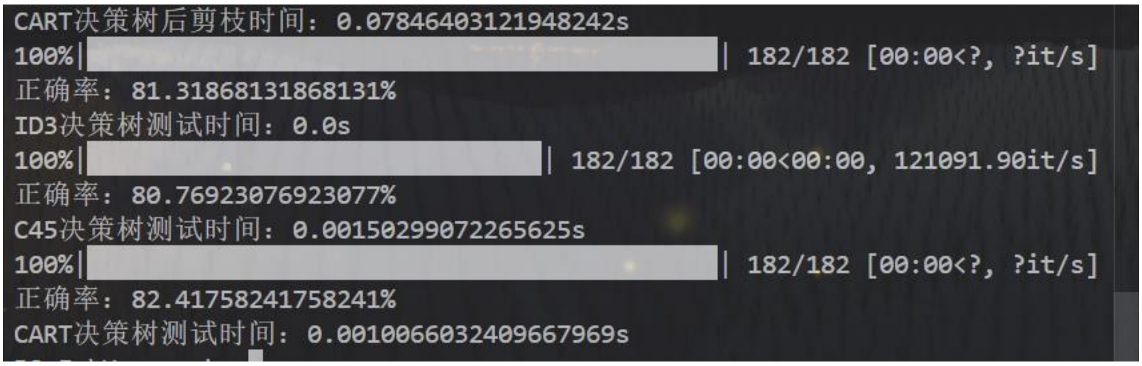
（4） 训练模型

（5） 测试结果正确率

（6） 模型可视化

3. 测试结果

正确率统计为：



```
CART决策树后剪枝时间：0.07846403121948242s
100%|██████████| 182/182 [00:00<?, ?it/s]
正确率：81.31868131868131%
ID3决策树测试时间：0.0s
100%|██████████| 182/182 [00:00<00:00, 121091.90it/s]
正确率：80.76923076923077%
C45决策树测试时间：0.0015029907226562 5s
100%|██████████| 182/182 [00:00<?, ?it/s]
正确率：82.41758241758241%
CART决策树测试时间：0.0010066032409667969s
```

决策树可视化为（顺序为 ID3 算法生成的决策树、C45 算法生成的决策树、CART 算法生成的决策树）：

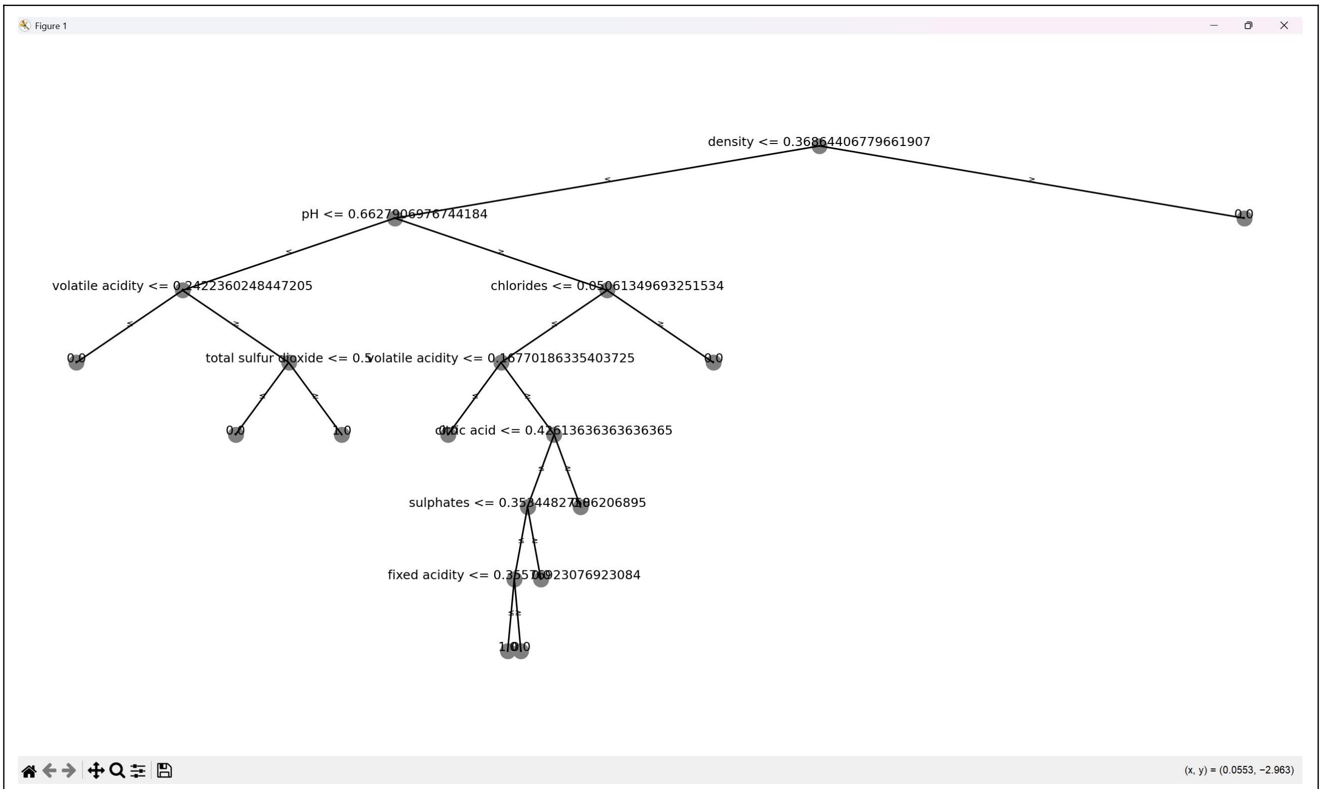


Figure 1
density <= 0.36864406779661907
pH <= 0.6627906976744184
volatile acidity <= 0.2422360248447205
chlorides <= 0.05061349693251534
total sulfur dioxide <= 0.5
volatile acidity <= 0.16770186335403725
citric acid <= 0.42613636363636365
0.0
1.0
(x, y) = (0.0553, −2.963)

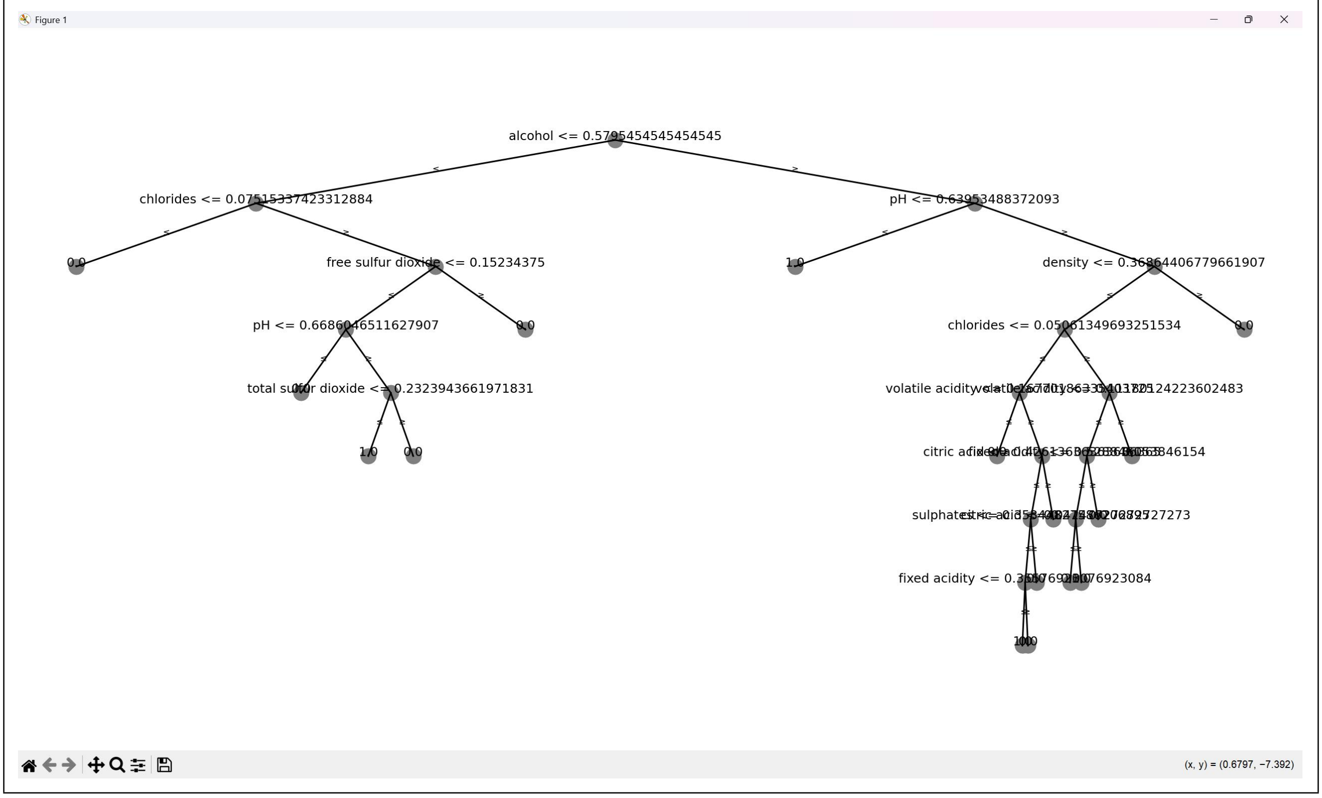
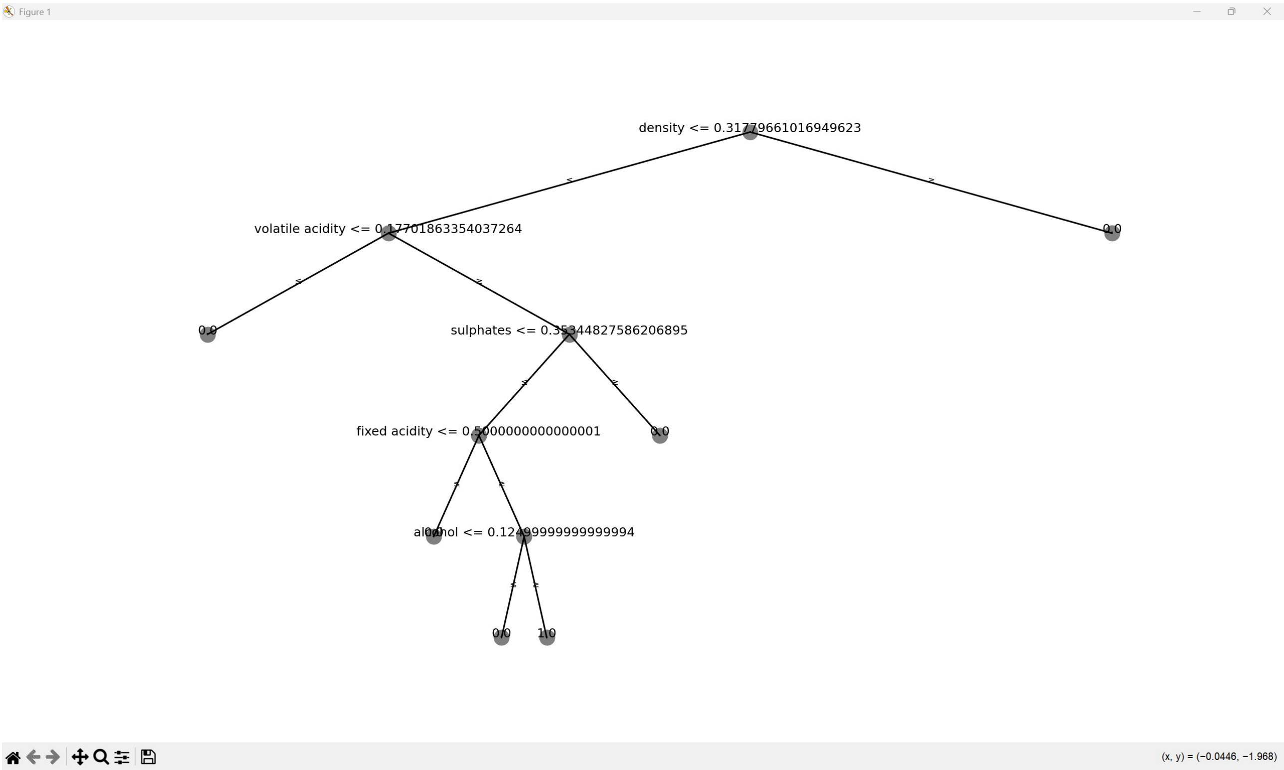


Figure 1
alcohol <= 0.5795454545454545
chlorides <= 0.07515337423312884
pH <= 0.63953488372093
free sulfur dioxide <= 0.15234375
density <= 0.36864406779661907
pH <= 0.6686046511627907
chlorides <= 0.05061349693251534
total sulfur dioxide <= 0.2323943661971831
0.0
1.0
(x, y) = (0.6797, −7.392)



附录：实现源代码

```python
'''超参数设置'''
train_set_rate = 0.7
'''训练集占比'''
max_depth = 10
'''决策树最大深度'''
def load_data(filename):
    import csv
    return csv.reader(open(filename, 'r'))
```

```python
class DataSet:
    """
    # DataSet
    这个类用于存储数据集，包括特征和标签，以及特征和标签的名称。
    ## method
    ### from_csv(filename)
    从 csv 文件中读取数据集，并返回一个 DataSet 对象。

    ### from_list(features, labels, features_names, labels_names)
    从列表中读取数据集，并返回一个 DataSet 对象。

    ### shuffle()
    随机打乱数据集。
    """
```

```python
    data_count : int
    '''数据个数'''
    feature_count : int
    '''特征个数'''
    features : list
    '''特征列表'''
    features_names : dict
    '''特征名称字典，用于从名称索引到列号'''
    labels : list
    '''标签列表'''
    labels_names : dict
    '''标签名称字典，用于从名称索引到列号'''
    labels_count : dict
    '''标签种类和对应个数'''
    def __init__(self, data_set = None, features:list = None, labels:list = None, features_names:dict = None, labels_names:dict = None):

        self.labels_names = {}
        self.features_names = {}
        self.labels_count = {}
        if data_set is not None:
            features_table = []
            labels_table = []

            for row in data_set:
                features_table.append(list(row[:len(row) - 1]))
                labels_table.append(list(row[-1]))
            labels_table[0] = [''.join(labels_table[0])]
            for i in range(len(labels_table[0])):
                self.labels_names[labels_table[0][i]] = i
            for i in range(len(features_table[0])):
                self.features_names[features_table[0][i]] = i
            self.labels = [[float(value) for value in row] for row in labels_table[1:]]
            self.features = [[float(value) for value in row] for row in features_table[1:]]
            self.data_count = len(self.features) - 1
            self.feature_count = len(self.features[0])
            for i in self.labels:
                if i[0] not in self.labels_count:
                    self.labels_count[i[0]] = 1
                else:
                    self.labels_count[i[0]] += 1
            return
        if data_set is None and features is not None and labels is not None and features_names is not None and labels_names is not None:
```

```
            self.features = features
            self.labels = labels
            self.features_names = features_names
            self.labels_names = labels_names
            if len(self.features) >= 1:

                self.data_count = len(self.features)
                self.feature_count = len(self.features[0])
            else:
                self.data_count = 0
                self.feature_count = 0
            for i in self.labels:
                if i[0] not in self.labels_count:
                    self.labels_count[i[0]] = 1
                else:
                    self.labels_count[i[0]] += 1
            return
        if data_set is None and features is None and labels is None and features_names is None and labels_names is None:
            self.data_count = 0
            self.feature_count = 0
            self.features = []
            self.labels = []
            self.features_names = {}
            self.labels_names = {}
            return
    @classmethod
    def from_csv(cls, filename):
        data_reader = load_data(filename)
        return cls(data_reader)
    @classmethod
    def from_list(cls, features:list, labels:list, features_names:dict, labels_names:dict):
        return cls(None, features, labels, features_names, labels_names)
    def shuffle(self):
        result = []
        for i in range(self.data_count):
            result.append(self.features[i] + self.labels[i])
        import random
        random.shuffle(result)

        self.features = [[float(value) for value in row[:len(row) - 1]] for row in result]
        self.labels = [[float(value) for value in row[len(row) - 1:]] for row in result]
    def min_max_normalize(self):
```

```
        '''
        最大最小值归一化处理
        '''
        min_values = [float('inf') for i in range(self.feature_count)]
        max_values = [float('-inf') for i in range(self.feature_count)]
        for i in range(self.data_count):
            for j in range(self.feature_count):
                if self.features[i][j] < min_values[j]:
                    min_values[j] = self.features[i][j]
                if self.features[i][j] > max_values[j]:
                    max_values[j] = self.features[i][j]
        for i in range(self.data_count):
            for j in range(self.feature_count):
                self.features[i][j] = (self.features[i][j] - 
min_values[j]) / (max_values[j] - min_values[j])

    def __repr__(self):
        return "DataSet(data_count={}, feature_count={}, 
features_names={}, labels_names={})".format(self.data_count, 
self.feature_count, self.features_names, self.labels_names) + "\n" + 
str(str(self.features[:10]) + "...") + "\n" + str(str(self.labels[:10]) + 
"...") + "\n" + str(self.labels_count)
```

```
    def information_entropy(self) -> float:
        '''
        计算信息熵
        '''
        entropy = 0
        for label in self.labels_count:
            import math
            p = self.labels_count[label] / self.data_count
            entropy -= p * math.log2(p)
        return entropy

    def gini_index(self) -> float:
        '''
        计算基尼指数
        '''
        gini = 1
        for label in self.labels_count:
            p = self.labels_count[label] / self.data_count
            gini -= p**2
        return gini
```

```
class DecisionNode:
    '''
```

```
    # DecisionNode
    决策树的节点类，包括特征名称、特征索引、阈值、子树、待分类样本集合。
    '''
    selectable_features : list
    '''可选择的特征'''
    feature_name : str
    feature_index : int
    threshold : float
    children : list
    divided_set : DataSet
    __leaf:bool
    __class:str

    def __init__(self, feature_name:str = None, feature_index:int = None,
threshold:float = None, children:list = None, divided_set:DataSet = None,
selectable_features:list = None, leaf:bool = False, class_label:str =
None):
        self.feature_name = feature_name
        self.feature_index = feature_index
        self.threshold = threshold
        self.children = children
        self.divided_set = divided_set
        self.selectable_features = selectable_features
        self.__leaf = False
```

```
        if class_label is not None:
            self.__class = class_label
        if self.divided_set is None:
            self.__leaf = True
        if leaf:
            self.__leaf = True
        if self.divided_set is not None:
            if self.divided_set.data_count == 0:
                self.__leaf = True
            if len([i for i in self.divided_set.labels_count.keys()]) ==
1:
                self.__leaf = True
        else:
            self.__leaf = True
        if self.__leaf is not None:
            if self.__leaf:
                if len([i for i in self.divided_set.labels_count.keys()])
== 1:
                    self.__class =
list(self.divided_set.labels_count.keys())[0]
        else:
```

```
            self.__class = None
        if leaf:
            self.__leaf = True
```

```
    def is_leaf(self):
        return self.__leaf
    def plot_text(self):
        if self.__leaf:
            return str(self.__class)
        else:
            return str(self.feature_name) + " <= " + str(self.threshold)
    def get_class(self):
        return self.__class
    def __divide(self, feature_index:int, threshold:float,
feature_name:str = None):
        '''
        划分子节点
        '''
        negative_features_list = []
        positive_features_list = []
        negative_labels_list = []
        positive_labels_list = []

        for i in range(self.divided_set.data_count):
            if self.divided_set.features[i][feature_index] <= threshold:
                negative_features_list.append(self.divided_set.features[
i])
                negative_labels_list.append(self.divided_set.labels[i])
            else:
                positive_features_list.append(self.divided_set.features[
i])
                positive_labels_list.append(self.divided_set.labels[i])
        positive_class = None
        negative_class = None
        positive_leaf = None
        negative_leaf = None

        if len(positive_labels_list) == 0:
            positive_class =
list(self.divided_set.labels_count.keys())[0]
            positive_leaf = True
        if len(negative_labels_list) == 0:
            negative_class =
list(self.divided_set.labels_count.keys())[1]
            negative_leaf = True
```

```
        positive_set = DataSet.from_list(positive_features_list,
positive_labels_list, self.divided_set.features_names,
self.divided_set.labels_names)
        negative_set = DataSet.from_list(negative_features_list,
negative_labels_list, self.divided_set.features_names,
self.divided_set.labels_names)
        selectable_features = [i for i in self.selectable_features]
        selectable_features.remove(feature_name)
```

```
        if positive_set.data_count == 0:
            if list(self.divided_set.labels_count.values())[0] >
list(self.divided_set.labels_count.values())[1]:
                positive_class =
list(self.divided_set.labels_count.keys())[0]
            else:
                positive_class =
list(self.divided_set.labels_count.keys())[1]
            positive_leaf = True
```

```
        if negative_set.data_count == 0:
            if list(self.divided_set.labels_count.values())[0] >
list(self.divided_set.labels_count.values())[1]:
                negative_class =
list(self.divided_set.labels_count.keys())[0]
            else:
                negative_class =
list(self.divided_set.labels_count.keys())[1]
            negative_leaf = True
        if selectable_features == []:
            if list(self.divided_set.labels_count.values())[0] >
list(self.divided_set.labels_count.values())[1]:
                negative_class =
list(self.divided_set.labels_count.keys())[0]
            else:
                negative_class =
list(self.divided_set.labels_count.keys())[1]
            negative_leaf = True
            if list(self.divided_set.labels_count.values())[0] >
list(self.divided_set.labels_count.values())[1]:
                positive_class =
list(self.divided_set.labels_count.keys())[0]
            else:
                positive_class =
list(self.divided_set.labels_count.keys())[1]
            positive_leaf = True
```

```
        self.children = [DecisionNode(None, None, None,None, positive_set,
selectable_features, positive_leaf, positive_class),DecisionNode(None,
None, None,None, negative_set, selectable_features, negative_leaf,
negative_class)]
        self.feature_name = feature_name
        self.feature_index = feature_index
        self.threshold = threshold

    def divide_ID3(self):
        '''
        ID3 算法划分子节点
        '''
        if self.__leaf:
            return
        # 先找出每一种特征的信息增益最大值和对应的阈值
        global_max_gain = {}
        global_max_threshold = {}
        from tqdm import tqdm
        for i in tqdm(self.selectable_features):
            max_gain = []
            max_threshold = []
            feature_index = self.divided_set.features_names[i]
            feature_values = [row[feature_index] for row in
self.divided_set.features]
            feature_values.sort()
            for j in tqdm(range(len(feature_values) - 1)):
                threshold = (feature_values[j] + feature_values[j + 1]) /
2
                self.__divide(feature_index, threshold, i)
                gain = self.information_gain()
                max_gain.append(gain)
                max_threshold.append(threshold)

            global_max_gain[i] = max(max_gain)
            for j in range(len(max_gain)):
                if max_gain[j] == global_max_gain[i]:
                    global_max_threshold[i] = max_threshold[j]
        # print(global_max_gain)
        # print(global_max_threshold)
        # 选出信息增益最大的特征和阈值
        max_gain_feature = max(global_max_gain, key=global_max_gain.get)
        max_gain_threshold = global_max_threshold[max_gain_feature]
        # print("选出特征:{}, 阈值:{}".format(max_gain_feature,
max_gain_threshold))
        self.__divide(train_set.features_names[max_gain_feature],
max_gain_threshold, max_gain_feature)
```

```
        self.children[0].divide_ID3()
        self.children[1].divide_ID3()
    def divide_C45(self):
        '''
        C4.5 算法划分子节点
        '''
        if self.__leaf:
            return
        # 先找出每一种特征的信息增益最大值和对应的阈值
        global_max_gain = {}
        global_max_threshold = {}
        from tqdm import tqdm
        for i in tqdm(self.selectable_features):
            max_gain = []
            max_threshold = []
            feature_index = self.divided_set.features_names[i]
            feature_values = [row[feature_index] for row in
self.divided_set.features]
            feature_values.sort()
            for j in tqdm(range(len(feature_values) - 1)):
                threshold = (feature_values[j] + feature_values[j + 1]) /
2
                self.__divide(feature_index, threshold, i)
                gain = self.information_gain()
                max_gain.append(gain)
                max_threshold.append(threshold)

            global_max_gain[i] = max(max_gain)
            for j in range(len(max_gain)):
                if max_gain[j] == global_max_gain[i]:
                    global_max_threshold[i] = max_threshold[j]

        avg_gain = sum(global_max_gain.values()) / len(global_max_gain) -
1e-10
        to_select_feature = [i for i in global_max_gain if
global_max_gain[i] >= avg_gain]
```

```
        max_ratio_gain = {}

        for i in tqdm(to_select_feature):
            self.__divide(self.divided_set.features_names[i],
global_max_threshold[i], i)
            gain_ration = self.gain_ratio()
            max_ratio_gain[i] = gain_ration
```

```
        # print(global_max_gain)
        # print(global_max_threshold)
        # 选出信息增益最大的特征和阈值
        try:
            max_gain_feature = max(max_ratio_gain,
key=max_ratio_gain.get)
        except Exception as e:
            print(e)
            print(global_max_gain)
            print(avg_gain)
            exit()
        max_gain_threshold = global_max_threshold[max_gain_feature]
        # print("选出特征:{}，阈值:{}".format(max_gain_feature,
max_gain_threshold))
        self.__divide(train_set.features_names[max_gain_feature],
max_gain_threshold, max_gain_feature)
        self.children[0].divide_C45()
        self.children[1].divide_C45()

    def divide_CART(self):
        '''
        CART 算法划分子节点
        '''
        if self.__leaf:
            return
        # 先找出每一种特征的基尼系数最小值和对应的阈值
        global_min_gini = {}
        global_min_threshold = {}
        from tqdm import tqdm
        for i in tqdm(self.selectable_features):
            min_gini = []
            min_threshold = []
            feature_index = self.divided_set.features_names[i]
            feature_values = [row[feature_index] for row in
self.divided_set.features]
            feature_values.sort()
            for j in tqdm(range(len(feature_values) - 1)):
                threshold = (feature_values[j] + feature_values[j + 1]) /
2

                self.__divide(feature_index, threshold, i)
                gini = self.gini_index()
                min_gini.append(gini)
                min_threshold.append(threshold)

            global_min_gini[i] = min(min_gini)
            for j in range(len(min_gini)):
```

```python
                if min_gini[j] == global_min_gini[i]:
                    global_min_threshold[i] = min_threshold[j]

        # print(global_min_gini)
        # print(global_min_threshold)
        # 选出基尼系数最小的特征和阈值
        min_gini_feature = min(global_min_gini, key=global_min_gini.get)
        min_gini_threshold = global_min_threshold[min_gini_feature]
        # print("选出特征:{}, 阈值:{}".format(min_gini_feature,
min_gini_threshold))
        self.__divide(train_set.features_names[min_gini_feature],
min_gini_threshold, min_gini_feature)
        self.children[0].divide_CART()
        self.children[1].divide_CART()

    def pruning(self):
        self.__leaf = True
        labels_count = self.divided_set.labels_count
        max_possible = max(labels_count.values())
        for i in labels_count:
            if labels_count[i] == max_possible:
                self.__class = i
                self.__leaf = True

    def unpruning(self):
        self.__leaf = False
        self.__class = None
```

```python
    def decision(self, data_dict:dict = None, data_list:list = None) -> str:
        '''
        进行决策获得结果
        '''
        if data_dict is not None:
            if self.__leaf:
                return self.__class
            if data_dict[self.feature_name] <= self.threshold:
                return self.children[0].decision(data_dict)
            else:
                return self.children[1].decision(data_dict)
        if data_list is not None:
            if self.__leaf:
                return self.__class
            if data_list[self.feature_index] <= self.threshold:
                return self.children[0].decision(None, data_list)
            else:
```

```
                return self.children[1].decision(None, data_list)
        if self.__leaf:
            return self.__class
    def information_gain(self) -> float:
        '''
        计算信息增益
        '''
        self_information_entropy =
self.divided_set.information_entropy()
        children_information_entropy = 0
        for child in self.children:
            coefficient = abs(child.divided_set.data_count /
self.divided_set.data_count)
            if coefficient == 0:
                continue
            children_information_entropy += coefficient *
child.divided_set.information_entropy()
        return self_information_entropy - children_information_entropy
    def gini_index(self) -> float:
        '''
        计算基尼指数
        '''
        gini_index = 0
        for child in self.children:
            coefficient = abs(child.divided_set.data_count /
self.divided_set.data_count)+1e-10
            gini_index+=coefficient * child.divided_set.gini_index()
        return gini_index

    def gain_ratio(self) -> float:
        '''
        计算信息增益比
        '''
        self_information_entropy =
self.divided_set.information_entropy()
        children_information_entropy = 0
        for child in self.children:
            import math
            coefficient = abs(child.divided_set.data_count /
self.divided_set.data_count)
            coefficient+=1e-10
            if coefficient == 0:
                continue
            children_information_entropy += coefficient *
child.divided_set.information_entropy() * math.log2(coefficient) + 1e-10
        children_information_entropy = -children_information_entropy
```

```
        return self_information_entropy / children_information_entropy
    def __repr__(self):
        return "DecisionNode(feature_name={}, feature_index={}, 
threshold={}, children={}, divided_set={})".format(self.feature_name, 
self.feature_index, self.threshold, self.children, self.divided_set)
import matplotlib.pyplot as plt
```

```
class PlotTreeNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None
from collections import deque
```

```
def array_to_bst(array):
    if not array:
        return None
```

```
    iter_array = iter(array)
    root = PlotTreeNode(next(iter_array))
    queue = deque([root])
```

```
    while queue:
        current_node = queue.popleft()
        try:
            left_value = next(iter_array)
            if left_value is not None:
                current_node.left = PlotTreeNode(left_value)
                queue.append(current_node.left)
            right_value = next(iter_array)
            if right_value is not None:
                current_node.right = PlotTreeNode(right_value)
                queue.append(current_node.right)
        except StopIteration:
            break
```

```
    return root
import matplotlib.pyplot as plt
```

```
def plot_tree(node, parent_name, node_name, edge_label, pos=None, x=0, y=0, 
layer=1):
    if pos is None:
        pos = {}
    pos[node_name] = (x, y)
    plt.text(x, y, str(node.val), fontsize=12, ha='center')
    plt.scatter(x, y, s=200, color='gray')
```

```
    if parent_name is not None:
        plt.plot([x, pos[parent_name][0]], [y, pos[parent_name][1]],
'k-')
        plt.scatter(x, y, s=200, color='gray')
        plt.text((x+pos[parent_name][0])/2, (y+pos[parent_name][1])/2,
edge_label, fontsize=8, ha='center')
    if node.left:
        plot_tree(node.left, node_name, node_name+"≤", '≤', pos,
x-1/2**layer, y-1, layer+1)
    if node.right:
        plot_tree(node.right, node_name, node_name+"≥", '≥', pos,
x+1/2**layer, y-1, layer+1)
    return pos
```

```
def draw_bst(root):
    fig, ax = plt.subplots()
    ax.axis('off')
    plot_tree(root, None, 'Root', None)
    plt.show()
```

```
class DecisionTree:
    '''
    # DecisionTree
    决策树类，包括根节点、最大深度。
    '''
    root : DecisionNode
    max_depth : int
    feature_names : dict
    label_names : dict
```

```
    def __init__(self, max_depth:int = None, root:DecisionNode = None ,
feature_names:dict = None, label_names:dict = None):
        if max_depth is not None:
            self.max_depth = max_depth
        if root is not None:
            self.root = root
        if feature_names is not None:
            self.feature_names = feature_names
        if label_names is not None:
            self.label_names = label_names
        pass
```

```
    @classmethod
    def train_ID3(cls, train_set:DataSet, max_depth:int):
        '''
        获取一个决策树对象
        '''
        features_names = [i for i in train_set.features_names.keys()]
        root = DecisionNode(None, None, None, None, train_set,
features_names)
        root.divide_ID3()
        return cls(max_depth, root, train_set.features_names,
train_set.labels_names)
    @classmethod
    def train_C45(cls, train_set:DataSet, max_depth:int):
        '''
        获取一个决策树对象
        '''
        features_names = [i for i in train_set.features_names.keys()]
        root = DecisionNode(None, None, None, None, train_set,
features_names)
        root.divide_C45()
        return cls(max_depth, root, train_set.features_names,
train_set.labels_names)
    @classmethod
    def train_CART(cls, train_set:DataSet, max_depth:int):
        '''
        获取一个决策树对象
        '''
        features_names = [i for i in train_set.features_names.keys()]
        root = DecisionNode(None, None, None, None, train_set,
features_names)
        root.divide_CART()
        return cls(max_depth, root, train_set.features_names,
train_set.labels_names)
    def __decision(self,data_list:list = None, data_dict:dict = None):
        '''
        进行决策获得结果
        '''
        return self.root.decision(data_dict, data_list)

    def decision_from_data_list(self, data_list:list):
        return self.__decision(data_list)
    def decision_from_data_dict(self, data_dict:dict):
        return self.__decision(None, data_dict)

    def test(self, test_set:DataSet, plot = None):
        '''
```

```
        测试模型
        '''
        correct_count = 0
        error_count = 0
        from tqdm import tqdm
        if plot is not None:
            for i in tqdm(range(test_set.data_count)):
                test_sample = {}
                for j in range(len(test_set.features[i])):
                    test_sample[list(test_set.features_names.keys())[j]]
= test_set.features[i][j]
                result = self.decision_from_data_dict(test_sample)
                # print("预测结果:{}, 真实结果:{}".format(result,
test_set.labels[i]))
                if result == test_set.labels[i][0]:
                    correct_count += 1
                else:
                    error_count += 1
            print("正确率:{}%".format(correct_count / (correct_count +
error_count) * 100))
        else:
            for i in tqdm(range(test_set.data_count)):
                test_sample = {}
                for j in range(len(test_set.features[i])):
                    test_sample[list(test_set.features_names.keys())[j]]
= test_set.features[i][j]
                result = self.decision_from_data_dict(test_sample)
                # print("预测结果:{}, 真实结果:{}".format(result,
test_set.labels[i]))
                if result == test_set.labels[i][0]:
                    correct_count += 1
                else:
                    error_count += 1
        return correct_count / (correct_count + error_count) * 100
    def __repr__(self):
        return "DecisionTree(max_depth={})".format(self.max_depth) + "\n"
+ str(self.root)

    def plot(self):
        '''
        绘制决策树
        '''
        node_array = [self.root.plot_text()]
        tem = self.root
        queue = deque([tem])
        while queue:
```

```
            node = queue.popleft()
            if node is None:
                continue
            if node.is_leaf():
                node_array.append(None)
                node_array.append(None)
                continue
            if node.children:
                if len(node.children) == 2:
                    node_array.append(node.children[0].plot_text())
                    node_array.append(node.children[1].plot_text())
                    queue.append(node.children[0])
                    queue.append(node.children[1])
                    # print("{} => {}".format(node.plot_text(),
node.children[0].plot_text()))
            else:
                node_array.append(None)
                node_array.append(None)
        print(node_array)
        # 示例使用
        root = array_to_bst(node_array)
        draw_bst(root)

    def post_pruning(self, test_set:DataSet):
        '''
        后剪枝
        '''
        tem = self.root
        queue = deque([tem])
        to_pruning = []
        while queue:
            node = queue.popleft()
            if node is None:
                continue
            to_pruning.append(node)
            if node.children:
                if len(node.children) == 2:
                    queue.append(node.children[0])
                    queue.append(node.children[1])
            else:
                continue
        to_pruning.reverse()
        for i in to_pruning:
            if i.is_leaf():
                continue
            old_correct_rate = self.test(test_set)
```

```
            i.pruning()
            new_correct_rate = self.test(test_set)
            if new_correct_rate < old_correct_rate:
                i.unpruning()
```

```
def random_split_data_set(dataset:DataSet, rate:float) ->
tuple[DataSet,DataSet]:
    '''
    训练集和测试机划分
    '''
    data_set_size = dataset.data_count
    data_set.shuffle()
    train_set = DataSet.from_list(dataset.features[:int(data_set_size *
rate)], dataset.labels[:int(data_set_size * rate)],
dataset.features_names, dataset.labels_names)
    test_set = DataSet.from_list(dataset.features[int(data_set_size *
rate):], dataset.labels[int(data_set_size * rate):],
dataset.features_names, dataset.labels_names)
    return train_set, test_set


if __name__ == '__main__':
    import time
    # 数据加载和预处理
    data_set = DataSet.from_csv('f:/Homework/机器学习作业/实验/实验
7/ex7Data/ex7Data.csv')
    # data_set = DataSet.from_csv('f:/Homework/机器学习作业/实验/实验
7/ex7Data/ex7DataSubset.csv')
    data_set.min_max_normalize()
    train_set, test_set = random_split_data_set(data_set, 0.7)

    # 训练
    start_time = time.time()
    ID3_tree = DecisionTree.train_ID3(train_set, 5)
    end_time = time.time()
    print("ID3 决策树训练时间:{}s".format(end_time - start_time))
```

```

    start_time = time.time()
    C45_tree = DecisionTree.train_C45(train_set, 5)
    end_time = time.time()
    print("C45 决策树训练时间:{}s".format(end_time - start_time))

    start_time = time.time()
```

```python
    CART_tree = DecisionTree.train_CART(train_set, 5)
    end_time = time.time()
    print("CART 决策树训练时间:{}s".format(end_time - start_time))
```

```python
    # 后剪枝
    start_time = time.time()
    ID3_tree.post_pruning(test_set)
    end_time = time.time()
    print("ID3 决策树后剪枝时间:{}s".format(end_time - start_time))

    start_time = time.time()
    C45_tree.post_pruning(test_set)
    end_time = time.time()
    print("C45 决策树后剪枝时间:{}s".format(end_time - start_time))

    start_time = time.time()
    CART_tree.post_pruning(test_set)
    end_time = time.time()
    print("CART 决策树后剪枝时间:{}s".format(end_time - start_time))
```

```python
    # 测试
    start_time = time.time()
    ID3_tree.test(test_set,1)
    end_time = time.time()
    print("ID3 决策树测试时间:{}s".format(end_time - start_time))

    start_time = time.time()
    C45_tree.test(test_set,1)
    end_time = time.time()
    print("C45 决策树测试时间:{}s".format(end_time - start_time))

    start_time = time.time()
    CART_tree.test(test_set,1)
    end_time = time.time()
    print("CART 决策树测试时间:{}s".format(end_time - start_time))

    # 可视化
    ID3_tree.plot()
    C45_tree.plot()
    CART_tree.plot()
```