

# 从零开始手写决策树(Decision Tree from Scratch)

## ——机器学习课程大作业

1950083 自动化 刘智宇

### 一、大作业资料链接

[项目地址 Github](<https://github.com/leizhenyu-lzy/BigHomework/tree/main/DecisionTree>)

[项目使用视频介绍 B 站](<https://www.bilibili.com/video/BV1yY4y1B7zr>)

### 二、项目文件夹结构说明

本次大作业主要依据《西瓜书》，从零开始实现了上面的各个算法。同时，使用面向对象思想，自己实现了两个类 `DatasetClass` 以及 `DecisionTreeNode` 类。同时，独立实现了各个相关的几个经典算法。

项目工程文件汇总及其说明如下图所示：

序号	文件/文件夹	说明
1	Datasets	存放数据集的文件夹 (包括四个西瓜数据集和一个iris数据集)
2	ReadmePics	存放readme.md所使用的图片
3	Rubbish	对本项目没有用到的文件， 以后迭代时可能会使用到
4	SavedDecisionTree	保存的决策树图片以及混淆矩阵图片
5	DatasetClass.py	数据集类
6	DecisionTreeNodeClass.py	决策树节点类
7	DecisionTreeVisualize.py	决策树可视化函数
8	GenerateDecisionTree.py	生成决策树函数
9	readme.md	本.md文件
10	UserToolFunction.py	存放一些通用函数和常量定义

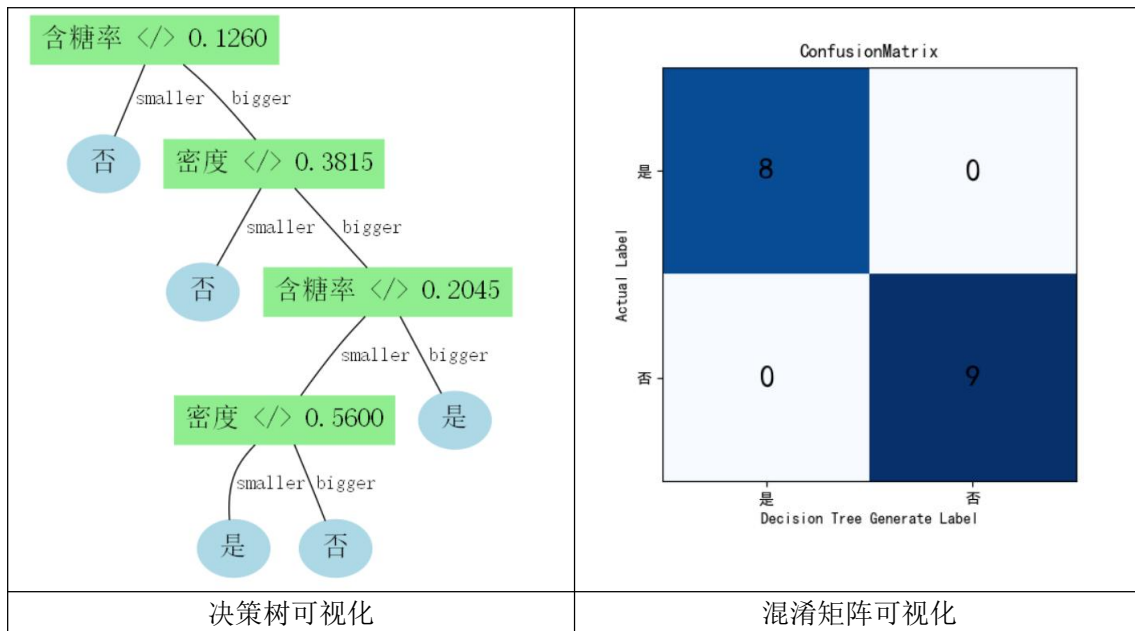
文件夹展示

文件说明

### 三、项目实现内容

将西瓜书中决策树章节几乎所有内容进行实现，包括但不限于以下几点：

- 1、决策树创建流程
- 2、各个划分选择算法
  - 1、信息增益方式
  - 2、信息增益率方式
  - 3、基尼系数方式
- 3、离散特征以及连续特征的处理
- 4、随机森林接口（方便后续拓展为集成学习中的随机森林）
- 5、可视化展示决策树及其表现（如下页所示）



个人认为本次自己实现的决策树的亮点：

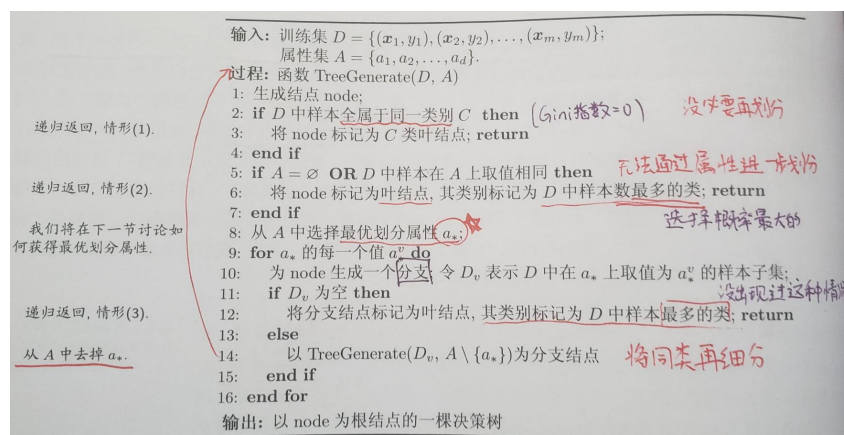
- 1、可以适配各种大小的数据集（不同的样本数，特征数）
- 2、可以适配不同数据类型的特征（整型、浮点型、字符串等等）（给数据集即可，不需要手动设定）
- 3、可以适配连续和离散特征（仅需指明连续特征即可）
- 4、决策树及其表现可视化

## 四、算法原理解理解

算法原理解理解我主要参考西瓜书进行学习，下面进行简述。

### （一）决策树建树流程

决策树（decision tree）是一类常见的机器学习方法，基于树结构进行决策，和人的处理机制类似（一系列的子决策）。决策树学习目的是为了产生一棵泛化能力强的决策树，基本原则遵循“分而治之”。生成决策树的过程就是一个递归的过程（如下图所示）：



## (二) 划分选择

### ①信息增益方式

信息熵用于度量样本集合纯度。信息增益则是信息熵变化量的度量。

$\text{Ent}(D) = - \sum_{k=1}^{ Y } p_k \log_2 p_k$	<p>“信息增益” (information gain)</p> $\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{ D^v }{ D } \text{Ent}(D^v)$ <p>加权平均 属性种类数</p>
信息熵	信息增益

信息增益越大越好。这一点其实很显然， $\text{Ent}(D)$ 大说明原来乱，而减数小说明划分后各个子集不乱，而这样差值大，因此信息增益要选择最大的。

经典的 ID3 决策树算法，就是使用信息增益为准则划分属性。

### ②信息增益率方式

信息增益率则是对于信息增益的一次改进：为了减少信息增益与准则对于可取值数目较多的属性的偏好，使用固有值 IV (intrinsic value) 对信息增益进行优化。很显然增益率也应该选择较大的。

<p>，增益率定义为</p> $\text{Gain\_ratio}(D, a) = \frac{\text{Gain}(D, a)}{\text{IV}(a)}$ <p>信息增益 固有值 某个属性可能取的值的个数</p> $\text{IV}(a) = - \sum_{v=1}^V \frac{ D^v }{ D } \log_2 \frac{ D^v }{ D }$	信息增益率以及固有值计算公式
--	----------------

### ③基尼指数

CART 决策树基于基尼指数 (Gini Index) 选择划分特征。在我看来，基尼系数和之前的信息增益类似。

$\begin{aligned} \text{Gini}(D) &= \sum_{k=1}^{ Y } \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{ Y } p_k^2 \end{aligned}$	$\text{Gini\_index}(D, a) = \sum_{v=1}^V \frac{ D^v }{ D } \text{Gini}(D^v)$
基尼值的定义	基尼指数的定义

值得注意的是，基尼指数与信息增益的减数部分类似，所以应选择使基尼指数较小的划分特征。

五、代码实现及其说明

代码主要分为 5 个.py 文件， 下面进行一一说明。（代码实现内容过多，就不在此进行展示，具体可在汇报开头的传送门或汇报结尾的附录进行查看）。

（一）DatasetClass.py

<p>DatasetClass 类 成员变量</p>	<pre>data dataset_name dataset_path features features_continuity features_name features_number features_possible_values label_name labels labels_possible_values samples_amount test_sub_dataset_list train_sub_dataset_list</pre>
<p>DatasetClass 类 成员函数</p>	<pre>__init__(self, dataset_path, dataset_name="MyDataset") comprehensiveInitializeDataset(self, continuity_list=None) readDataset(self, header=0, index_col=None) printDatasetInfo(self) countFeaturesPossibleValues(self) showFeaturesPossibleValues(self) countLabelsPossibleValues(self) countLabelsPossibleValuesByList(self, samples_list=None) setFeaturesContinuity(self, continuity_list=None) splitSamplesList(self, division_feature_id, samples_list=None, continuity_split_value=0) computeInformationEntropy(self, samples_list=None) computeOneFeatureEntropyGain(self, division_feature_id, samples_list=None) selectDivisionFeatureByEntropyGain(self, available_division_feature_list=None, samples_list=None) computeIntrinsicValue(self, division_feature_id, samples_list=None) computeOneFeatureGainRatio(self, division_feature_id, samples_list=None) selectDivisionFeatureByGainRatio(self, available_division_feature_list=None, samples_list=None) computeGiniValue(self, samples_list=None) computeOneFeatureGiniIndex(self, division_feature_id, samples_list=None) selectDivisionFeatureByGiniIndex(self, available_division_feature_list=None, samples_list=None) getDivisionFeature(self, available_division_feature_list=None, samples_list=None, get_division_feature_method=user.entropy_gain) judgeSamplesHaveSameFeatures(self, features_list, samples_list) getLabelsPossibleValuesList(self) seperateTrainSetAndTestSet(self, train_dataset_numbers=0)</pre>

重要成员函数说明：

- 1.comprehensiveInitializeDataset：通过读取数据集对当前数据集进行较为全面初始化
- 2.readDataset：读取数据集，并添加一些额外信息
- 3.printDatasetInfo：打印数据基本信息
- 4.countLabelsPossibleValuesByList：统计给定样本类别的标签分布
- 5.setFeaturesContinuity：通过传入列表设置各个特征的连续性
- 6.splitSamplesList：通过给定的特征对样本列表进行划分（对于连续和离散都适用）
- 7.selectDivisionFeatureByEntropyGain：通过信息增益的方式选出最合适的划分特征
- 8.selectDivisionFeatureByGainRatio：通过信息增益率的方式选出最合适的划分特征
- 9.selectDivisionFeatureByGiniIndex：通过基尼系数的方式选出最合适的划分特征
- 10.getDivisionFeature：将上述三个选择统一在一起作为对外的接口
- 11.judgeSamplesHaveSameFeatures：判断给定的 samples 再给定的 features 十分完全相同（方便后续判断决策树是否跳出循环）
- 12.seperateTrainSetAndTestSet：拆分数集和训练集

## (二) DecisionTreeNodeClass.py

### DecisionTreeNodeClass 类成员变量&类成员函数

```
m __init__(self)
m getNodeInfo(self)
m attachChildNode(self, childnode, childnode_feature_value)
m updateDivisionProperty(self, div_feature)
m printDecisionTreeDepthFirst(self)
m printDecisionTreeBreadthFirst(self)
m setIncludeSample(self, include_sample_list)
m setAvailableFeaturesId(self, available_feature_id_list)
m saveDecisionTreeToFile(self, filename="DecisionTree.txt")
m readDecisionTreeFromFile(self, filename="DecisionTree.txt")
m isLeaf(self)
m getSampleLabelByDecisionTree(self, sample_features_list, features_continuity_list)
m getConfusionMatrixDict(self, dataset, test_list=None)
f available_features_id
f childnode_division_feature_values
f childnodes_id_list
f childnodes_list
f continuous_feature_split_value
f division_feature_id
f final_label
f global_node_id
f include_samples
f node_id
f parentnode
f parentnode_id
```

除此以外，我还定义了一个类共享的变量：**global\_node\_id**，在每次创建新节点时，会进行自增，方便对节点进行编号和后续绘图。

**global\_node\_id = 0** # 类共享变量（在有新 **treenode** 创建时会自增）。

重要成员函数说明：

- 1.**getNodeInfo**：打印节点信息
- 2.**attachChildNode**：将子节点添加到当前节点的子节点列表（进行链接）
- 3.**printDecisionTreeDepthFirst**：深度优先在终端打印决策树基本信息
- 4.**printDecisionTreeBreadthFirst**：广度优先在终端打印决策树基本信息
- 5.**isLeaf**：判断当前节点是否为叶节点
- 6.**getSampleLabelByDecisionTree**：获取决策树根据已知特征进行的 **label** 判断结果
- 7.**getConfusionMatrixDict**：生成字典形式的混淆矩阵

### （三）UserToolFunction.py

本文件开头定义了一些数据集和决策树会使用到的一些全局变量。后面实现了一些很简单的函数，我认为不适合放在其他位置，故放到了此处。

UserToolFunction 相关变量以及函数
<div><div>v</div>dataset_header</div> <div><div>v</div>dataset_index_col</div> <div><div>v</div>DatasetsFolder</div> <div><div>v</div>DatasetName</div> <div><div>v</div>DatasetPath</div> <div><div>v</div>DatasetContinuityList</div> <div><div>v</div>entropy_gain</div> <div><div>v</div>gain_ratio</div> <div><div>v</div>gini_index</div> <div><div>v</div>get_division_feature_method</div> <div><div>v</div>get_division_feature_methods_name</div> <div><div>v</div>viz_decision_tree_name</div> <div><div>f</div>printSeparateBar(length=40, sign="*")</div> <div><div>f</div>getMaxValueIndex(sequence)</div> <div><div>f</div>pLog2p(possibility)</div>

### （四）GenerateDecisionTree.py

GenerateDecisionTree 相关变量以及函数
<div><div>f</div>generateDecisionTree(dataset, current_node, get_division_feature_method)</div>

本文件较为简单，仅有一个生成决策树的函数（递归思想），该函数按照西瓜书图 4.2 的结构进行编写。

在该文件的\_\_main\_\_函数中，我将整个决策树生成即可视化进行串联。即可以运行该 main 函数根据给定数据集训练出相应决策树。（注意，由于西瓜数据集过于简单，我并没有进行 train、test 的拆分，我另外添加了一个鸢尾花数据集（IRIS）进行了拆分，并进行检验。）



## (五) DecisionTreeVisualize.py

### DecisionTreeVisualize 相关变量以及函数

```
m __init__(self)
m getNodeInfo(self)
m attachChildNode(self, childnode, childnode_feature_value)
m updateDivisionProperty(self, div_feature)
m printDecisionTreeDepthFirst(self)
m printDecisionTreeBreadthFirst(self)
m setIncludeSample(self, include_sample_list)
m setAvailableFeaturesId(self, available_feature_id_list)
m saveDecisionTreeToFile(self, filename="DecisionTree.txt")
m readDecisionTreeFromFile(self, filename="DecisionTree.txt")
m isLeaf(self)
m getSampleLabelByDecisionTree(self, sample_features_list, features_continuity_list)
m getConfusionMatrixDict(self, dataset, test_list=None)
f available_features_id
f childnode_division_feature_values
f childnodes_id_list
f childnodes_list
f continuous_feature_split_value
f division_feature_id
f final_label
f global_node_id
f include_samples
f node_id
f parentnode
f parentnode_id
```

[Graphviz 官方文档](<https://graphviz.readthedocs.io/en/stable/examples.html>)

这一部分调用了 **Graphviz** 包进行决策树的可视化及保存以及 **matplotlib** 包进行混淆矩阵的绘制及保存。其中 **Graphviz** 包的使用介绍略微有些难找，故将其网址放在下方。

在文件最前面定义了一些常量，主要用于设置决策树绘制的颜色字体等，字体最好不要改动，否则可能会导致中文乱码。

六、项目使用说明

首先将自己的数据集放入 Datasets 文件夹中（也可以不放，将后面的 DatasetPath 进行相应修改即可）。

其次在 UserToolFunction.py 文件中，填入以下内容：

- 1. DatasetName、DatasetPath 两个数据集基本信息
- 2. 对于有连续特征的数据集，还需额外显式指明 DatasetContinuityList，用 0 和 1 分别表示每个特征的连续性（0 为非连续，1 为连续），DatasetContinuityList 长度应 等于 特征的个数，且对应每一个特征。对于全为离散特征的数据集，写 None 即可。
- 3. 对于想要拆分 train 和 test 还需另外传入一个 TrainSamplesAmount 指明希望拆分得到的训练集个数。
- 4. 指明使用的选取最优划分特征的方法：get\_division\_feature\_method（对于有连 续特征的数据集目前只能选择 entropy\_gain）
- 5. 保存决策树结构图片和混淆矩阵图片的文件名（有默认值）

另外对于生产决策树过程，对 GenerateDecisionTree.py 中的 main 函数进行选取

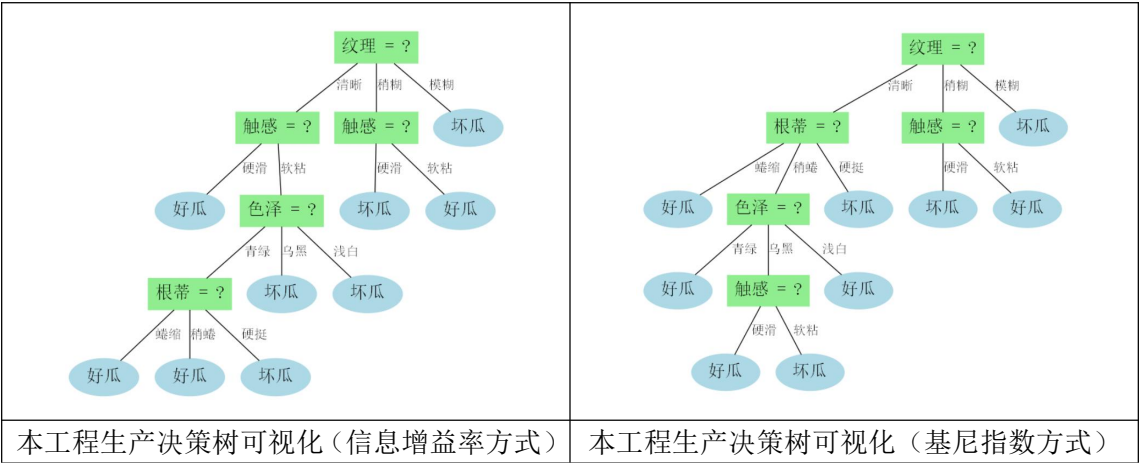
- 1. 对于不想划分训练集和测试集：使用上方类似于西瓜数据集部分
  - 2. 对于想要划分训练集和测试集：使用下方类似鸢尾花数据集部分
- 最后运行 GenerateDecisionTree.py 的 main 函数即可。

七、代码测试结果

（一）西瓜数据集 2.0（只有 6 个离散特征）

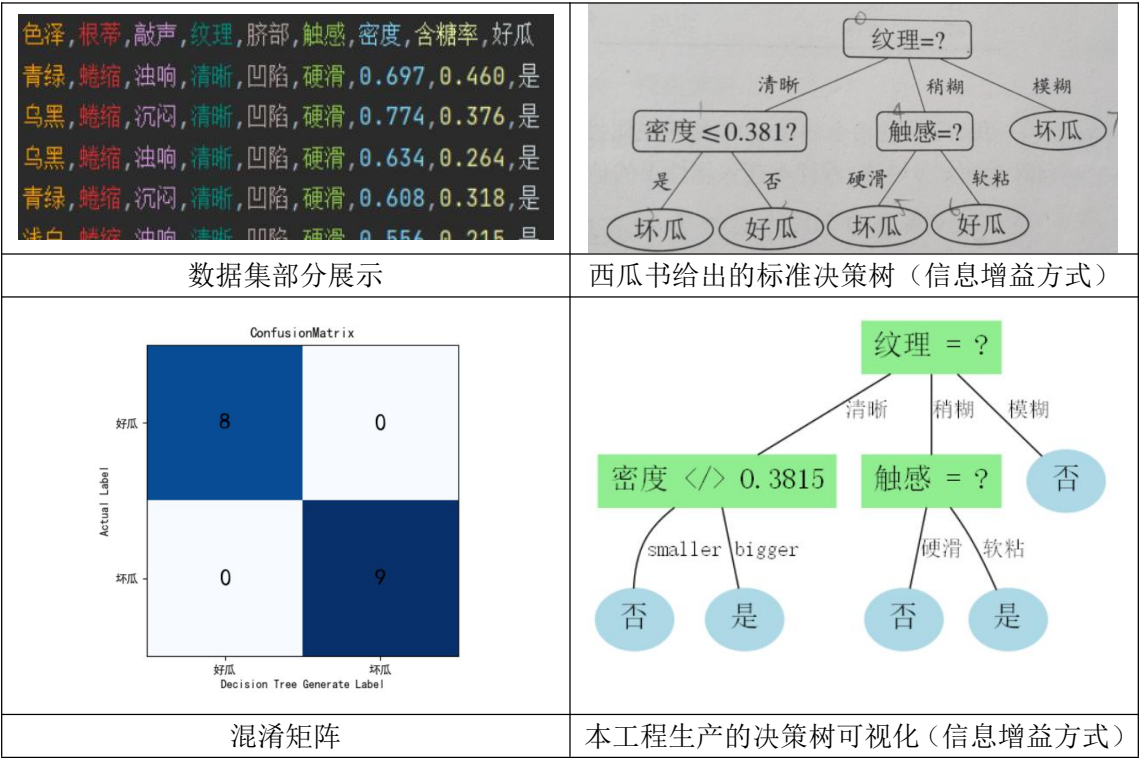
数据集部分展示	西瓜书给出的标准决策树（信息增益方式）
混淆矩阵	本工程生产决策树可视化（信息增益方式）






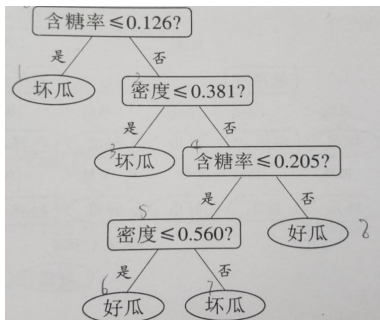
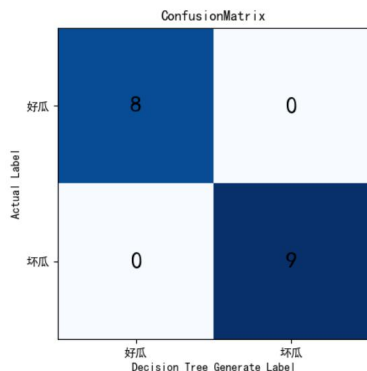
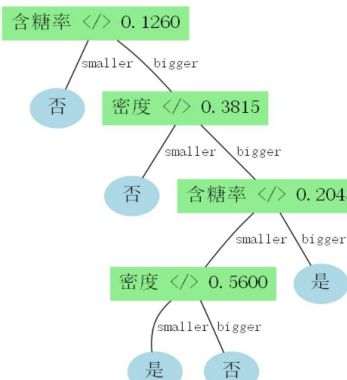
很显然，对于仅含有离散特征的数据集，本工程生成的决策树和《西瓜书》，上给出的完全一致，证明自己实现的决策树算法的正确性。

（二）西瓜数据集 3.0（有 2 个连续特征和 6 个离散特征）



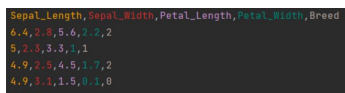
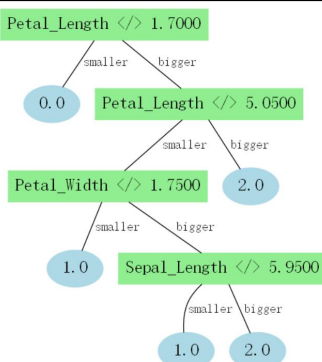
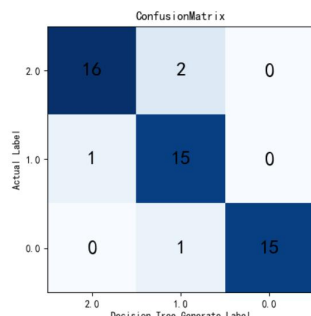
很显然，对于同时含有离散特征和连续特征的数据集，本工程生成的决策树和《西瓜书》，上给出的完全一致，进一步证明自己实现的决策树算法的正确性。

### (三) 西瓜数据集 3.0alpha (只有 2 个连续特征)

	
数据集部分展示	西瓜书给出的标准决策树（信息增益方式）
	
混淆矩阵	本工程生产的决策树可视化（信息增益方式）

很显然,对于仅含有离散特征的数据集,本工程生成的决策树和《西瓜书》,上给出的完全一致,进一步证明自己实现的决策树算法的正确性。

### (四) 鸢尾花数据集

		
数据集部分展示	本工程生产的决策树可视化 (信息增益方式)	混淆矩阵

数据集由四个连续特征组成,使用留出法划分数据集 (70 个样本) 和测试集 (50 个样本),生成的决策树如下所示 (由于是随机选取的样本分给 train 和 test 所以每次结果未必相同)。决策树上的 0.0、1.0、2.0 节点分别对应三种鸢尾花。

## 八、学习体会

决策树算法虽然看上去十分的浅显易懂,但是在实际进行代码实现时却没有想象中那么简单。许多时候以为自己理解某个部分了,但是在代码实现时发现“南辕北辙”,才意识到自己的问题,所以代码实现有时是很有必要的。

此外,在写代码时常常由于思绪混乱导致遇到各种 **bug**,好在最终都一一解决了,在调试代码的过程中也锻炼了我的耐心。

本次决策树代码实现使用了面向对象思想(树节点类、数据集类),但我在一开始却没有使用该思想。通过两个思想的对比也让我意识到了面向对象的优点,体会到了封装后在使用上的便捷。

本工程实现的决策树仅仅复现了西瓜书上的大部分代码以及一些自己感兴趣的可视化代码。书上还有一些决策树的基础功能有待实现:剪枝(预剪枝、后剪枝)等等。今后有时会将会将书上内容全部完成,同时将该工程拓展为随机森林。

我还设想过将决策树以文件的形式进行存储,这样不必每次重新训练,仅需从保存的文件读取即可,但是该功能尚未实现,后续我会选择较为优雅的方式对其进行实现。

当然,除了决策树,我还会在闲暇之余实现其他的机器学习算法,为今后的深入学习打下基础。

## 九、附录（代码实现）

### （一）DatasetClass.py

```
# 1950083 自动化 刘智宇

# Thirdparty
import pandas as pd
import random

# User
import UserToolFunction as user

"""
Dataset 类
"""
class Dataset:
    # __init__
    def __init__(self, dataset_path, dataset_name="MyDataset"):
        self.dataset_name = dataset_name # 数据集名称
        self.dataset_path = dataset_path # 数据集文件路径
        self.samples_amount = 0 # 数据集样本个数
        self.data = [] # 数据集内容
        self.features_number = 0 # 特征索引个数
        self.features_name = [] # 全部特征名称, 用于后续决策树划分
        self.features = [] # 数据集的全部样本特征数据
        self.features_continuity = [] # 连续特征标记 (1 表示连续, 0 表示离散)
        self.labels = []
        self.label_name = "" # 标签索引名称
        self.features_possible_values = [] # 各个特征的全部可能的特征值, 列表元素为字典, 字典内为
        # feature 的值及对应个数
        self.labels_possible_values = {} # 标签的可能值 (字典, labels 可能值和对应数量)
        self.train_sub_dataset_list = []
        self.test_sub_dataset_list = []
        # self.available_features_number_list = [] # not const
    # 高级初始化
    def comprehensiveInitializeDataset(self, continuity_list=None):
        self.readDataset(header=user.dataset_header, index_col=user.dataset_index_col)
        self.countFeaturesPossibleValues() #
        self.countLabelsPossibleValues() #
        self.setFeaturesContinuity(continuity_list) # 确定是否为连续特征(pending)
    # 读取数据集
    def readDataset(self, header=0, index_col=None):
        data_frame = pd.read_csv(filepath_or_buffer=self.dataset_path, header=header,
        index_col=index_col) # 读取文件到dataframe
        # print(data_frame.dtypes)
        self.data = data_frame.values # 数据转到列表中
```

```

data_rows, data_cols = self.data.shape
self.samples_amount = data_rows
self.features = self.data[:, 0:-1]
self.features_number = data_cols - 1 # 认为标签只有1列
self.labels = self.data[:, -1]

dataset_index_name = data_frame.columns.values # 取得数据各项索引名称
self.features_name = dataset_index_name[0:-1]
self.label_name = dataset_index_name[-1]

# self.available_features_number_list = [i for i in range(self.features_number)] # 还没有被决策树用到的特征编号

# 打印数据集信息
def printDatasetInfo(self):
    user.printSeparateBar()
    print("dataset_name: ", self.dataset_name, "\n")
    print("dataset_path: ", self.dataset_path, "\n")
    print("sample_numbers: ", self.samples_amount, "\n")
    # print("data:\n", self.data, "\n")
    print("features_number: ", self.features_number, "\n")
    print("features_name:\n", self.features_name, "\n")
    # print("features:\n", self.features, "\n")
    print("features_continuity:\n", self.features_continuity, "\n")
    print("label_name:\n", self.label_name, "\n")
    # print("labels:\n", self.labels, "\n")
    print("features_possible_values:\n", self.features_possible_values, "\n")
    print("labels_possible_values:\n", self.labels_possible_values, "\n")
    print("label_name:\n", self.label_name, "\n")
    user.printSeparateBar()

# 统计每一个特征中不重复的值的种类, 对应于多元特征
def countFeaturesPossibleValues(self):
    for col in range(self.features_number):
        temp_value_dict = {self.features[0][col]: 1}
        for row in range(1, self.samples_amount):
            new_value_flag = 1 # 判断是否为新值, 默认为新值
            current_value = self.features[row][col] # 当前处理的特征的具体值
            for value in temp_value_dict:
                if current_value == value: # 之前出现过该特征值
                    new_value_flag = 0
                    temp_value_dict[value] += 1
                    break
            if new_value_flag == 1: # 没有出现过重复特征值, 将其添加
                temp_value_dict[current_value] = 1
            self.features_possible_values.append(temp_value_dict)

# 输出self.all_features_all_values
def showFeaturesPossibleValues(self):

```

```

        for cnt in range(self.features_number):
            print(self.features_name[cnt], "\t : \t", self.features_possible_values[cnt])
# 统计整个数据集的标签值种类及其个数
def countLabelsPossibleValues(self):
    labels_possible_values = self.countLabelsPossibleValuesByList()
    self.labels_possible_values = labels_possible_values
# 统计给定编号的数据的标签值(Label)的种类及其个数
def countLabelsPossibleValuesByList(self, samples_list=None):
    if samples_list is None:
        samples_list = [cnt for cnt in range(self.samples_amount)]
    if len(samples_list) == 0: # 如果出现划分属性最大值有多个, 会导致 bigger 部分没有值
        return {}
    # print(samples_list)
    labels_possible_values_dict = {self.labels[samples_list[0]]: 1}
    for sample_number in samples_list[1:]: # 遍历给定样本列表的 Label
        new_label_value_flag = 1
        temp_label_value = self.labels[sample_number]
        for exist_value in labels_possible_values_dict:
            if temp_label_value == exist_value:
                new_label_value_flag = 0
                labels_possible_values_dict[temp_label_value] += 1
                break
        if new_label_value_flag == 1:
            labels_possible_values_dict[temp_label_value] = 1
    return labels_possible_values_dict
# # 判断是否所有样本的 Label 都相同
# def judgeSamplesBelongSameCategory(self, samples_list=None):
#     label_possible_values = self.countLabelsPossibleValuesByList(samples_list)
#     if len(label_possible_values) == 1:
#         # print("True ", list(label_possible_values.keys()))
#         return True
#     else:
#         # print("False")
#         return False
# 判断特征的连续性
def setFeaturesContinuity(self, continuity_list=None):
    if continuity_list is None or len(continuity_list) == 0: # 没有传入连续值列表, 默认全部不
连续
        self.features_continuity = [0] * self.features_number
        return True
    elif len(continuity_list) == self.features_number:
        self.features_continuity = continuity_list
        return True
    else: # 传入 continuity_list 但是长度对不上, 说明 list 长度错误, 打印错误信息

```



```

        print("[INFO] : The length of continuity_list is wrong.")
        return False

# 通过给定特征拆分给定样本列表（需考虑是否为连续特征）
def splitSamplesList(self, division_feature_id, samples_list=None,
continuity_split_value=0):
    if samples_list is None: # 没有给定 samples_list, 就统计所有值
        samples_list = [cnt for cnt in range(self.samples_amount)]
    split_dict = {}
    if self.features_continuity[division_feature_id]: # 连续特征
        # print("splitSampleList : Continue")
        split_dict['bigger'] = []
        split_dict['smaller'] = []
        # 根据西瓜书：大于的归为一类，不大于的归为一类
        for sample_id in samples_list:
            if self.features[sample_id][division_feature_id] > continuity_split_value:
                split_dict['bigger'].append(sample_id)
            else:
                split_dict['smaller'].append(sample_id)
    else: # 非连续特征
        # print("splitSampleList : Discrete")
        for division_feature_value in
list(self.features_possible_values[division_feature_id].keys()):
            split_dict[division_feature_value] = []
            for sample_number in samples_list:
                split_dict[self.features[sample_number][division_feature_id]].append(sample_num
ber)

            for division_feature_value in
list(self.features_possible_values[division_feature_id].keys()): # 避免空的列表返回
                if len(split_dict[division_feature_value]) == 0:
                    split_dict.pop(division_feature_value) # 删除值为空列表的键值对
        return split_dict

# 给定样本列表，计算信息熵（对于是否连续都适用，只统计 label）
def computeInformationEntropy(self, samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]
    information_entropy = 0.0 # 信息熵
    label_values_dict = self.countLabelsPossibleValuesByList(samples_list) # 统计标签值的字典

    for label_value in label_values_dict:
        label_proportion = (0.0 + label_values_dict[label_value])/len(samples_list)
        information_entropy -= user.pLog2p(label_proportion)

    # print(information_entropy)
    return information_entropy

# 计算给定的划分特征、样本列表的信息增益（需考虑是否为连续特征）

```

```

def computeOneFeatureEntropyGain(self, division_feature_id, samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]
    # 先计算整体的信息熵
    overall_information_entropy = self.computeInformationEntropy(samples_list)
    entropy_gain = overall_information_entropy
    # print('division_feature_id: ', division_feature_id)
    if self.features_continuity[division_feature_id]: # 连续特征
        feature_values_list = []
        for sample_id in samples_list:
            feature_values_list.append(self.features[sample_id][division_feature_id])
        feature_values_list.sort()
        # print("Sorted Value List : ", feature_values_list)
        best_entropy_gain_split_value = 0.0 # 最大信息增益对应的特征划分值
        max_entropy_gain = 0.0 # 最大的信息增益
        for value_id in [i+1 for i in range(len(feature_values_list)-1)]: # 将所有候选划分点
            # 加入待测列表
            temp_entropy_gain = entropy_gain # 每次循环更新原值
            continuity_split_value =
            (feature_values_list[value_id]+feature_values_list[value_id-1])/2
            split_dict = self.splitSamplesList(division_feature_id, samples_list,
            continuity_split_value)
            for division_feature_value in split_dict:
                sub_information_entropy =
                self.computeInformationEntropy(split_dict[division_feature_value]) # 计算划分后的子样本集的信息
                # 熵
                sub_information_entropy_weight = len(split_dict[division_feature_value]) /
                len(samples_list) # 计算子样本集相应权重 (子样本集的占比)
                temp_entropy_gain -= sub_information_entropy * sub_information_entropy_weight
                # print("%2d: temp_entropy_gain: %.4f" % (value_id, temp_entropy_gain))
                # print("%2d: continuity_split_value: %.4f" % (value_id, continuity_split_value))
                if temp_entropy_gain > max_entropy_gain: # 更新连续特征的最优划分值, 更新最大的信
                    # 息增益
                    best_entropy_gain_split_value = continuity_split_value
                    max_entropy_gain = temp_entropy_gain
            return max_entropy_gain, best_entropy_gain_split_value
        else: # 非连续特征
            # 对样本列表根据划分特征进行拆分, 得到字典 (字典中的 value 为列表)
            split_dict = self.splitSamplesList(division_feature_id, samples_list)
            # 计算拆分后各样本子列表的加权信息熵, 并从信息增益中减去
            for division_feature_value in split_dict:
                sub_information_entropy =
                self.computeInformationEntropy(split_dict[division_feature_value]) # 计算划分后的子样本集的信息
                # 熵

```

```

        sub_information_entropy_weight =
len(split_dict[division_feature_value])/len(samples_list) # 计算子样本集相应权重（子样本集的占比）
        entropy_gain -= sub_information_entropy * sub_information_entropy_weight

        return entropy_gain, None

# 通过信息增益的方式选出最合适的划分特征
def selectDivisionFeatureByEntropyGain(self, available_division_feature_list=None,
samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]

    if available_division_feature_list is None: # 没有给定 available_division_feature_list,
认为全部可用
        available_division_feature_list = [cnt for cnt in range(self.features_number)]

    chosen_division_feature_id = None
    max_entropy_gain = 0.0
    best_split_value = None

    for division_feature_id in available_division_feature_list:
        temp_entropy_gain, split_value =
self.computeOneFeatureEntropyGain(division_feature_id, samples_list)

        if temp_entropy_gain > max_entropy_gain:
            chosen_division_feature_id = division_feature_id
            max_entropy_gain = temp_entropy_gain
            best_split_value = split_value # 对于连续特征特有的最佳划分特征

    print(chosen_division_feature_id, best_split_value)
    return chosen_division_feature_id, best_split_value

# 计算某一特征的固有价值 intrinsic value
def computeIntrinsicValue(self, division_feature_id, samples_list=None):
    intrinsic_value = 0.0

    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]

    split_dict = self.splitSamplesList(division_feature_id, samples_list)
    for division_feature in split_dict:
        sub_samples_proportion = len(split_dict[division_feature])/len(samples_list)
        intrinsic_value -= user.pLog2p(sub_samples_proportion)

    # print("intrinsic_value : ", intrinsic_value)
    return intrinsic_value

# 计算某一特征的信息增益率
def computeOneFeatureGainRatio(self, division_feature_id, samples_list=None):
    feature_intrinsic_value = self.computeIntrinsicValue(division_feature_id,
samples_list) # 计算特征的固有价值

    feature_information_gain = self.computeOneFeatureEntropyGain(division_feature_id,
samples_list)[0] # 计算特征的信息增益

    return feature_information_gain/feature_intrinsic_value # 计算特征增益率

# 通过信息增益率的方式选出最合适的划分特征

```

```

def selectDivisionFeatureByGainRatio(self, available_division_feature_list=None,
samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]
    if available_division_feature_list is None: # 没有给定 available_division_feature_list,
认为全部可用
        available_division_feature_list = [cnt for cnt in range(self.features_number)]
    chosen_division_feature_id = 0
    max_gain_ratio = 0.0
    for division_feature_id in available_division_feature_list:
        temp_gain_ratio = self.computeOneFeatureGainRatio(division_feature_id, samples_list)
        # print(temp_gain_ratio)
        if temp_gain_ratio > max_gain_ratio:
            chosen_division_feature_id = division_feature_id
            max_gain_ratio = temp_gain_ratio
    return chosen_division_feature_id, None

# 计算基尼值
def computeGiniValue(self, samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部样本
        samples_list = [cnt for cnt in range(self.samples_amount)]
    gini_value = 1.0 # 使用  $1 - [\sigma(p)]^2$  的公式进行基尼值的度量, 故从 1.0 开始减
    label_values_dict = self.countLabelsPossibleValuesByList(samples_list) # 统计标签值的字典
    for label_value in label_values_dict:
        label_proportion = (0.0 + label_values_dict[label_value])/len(samples_list)
        gini_value -= pow(label_proportion, 2)
    # print("gini_value : ", gini_value)
    return gini_value

# 计算某一划分特征的基尼指数 (基尼指数不等同于基尼值, 多进行了加权平均)
def computeOneFeatureGiniIndex(self, division_feature_id, samples_list=None):
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]
    gini_index = 0.0
    # 对样本列表根据划分特征进行拆分, 得到字典 (字典中的 value 为样本 id 列表)
    split_dict = self.splitSamplesList(division_feature_id, samples_list)
    # 计算拆分后各样本子列表的加权信息熵, 并从信息增益中减去
    for division_feature_id in split_dict:
        sub_gini_value = self.computeGiniValue(split_dict[division_feature_id]) # 计算划分
后的子样本集的信息熵
        sub_gini_value_weight = len(split_dict[division_feature_id]) / len(samples_list) #
计算子样本集相应权重 (子样本集的占比)
        gini_index += sub_gini_value * sub_gini_value_weight
    # print("gini_index : ", gini_index)
    return gini_index

```

```

# 通过基尼系数的方式选出最合适的划分特征
def selectDivisionFeatureByGiniIndex(self, available_division_feature_list=None,
samples_list=None):
    # 注意根据基尼系数选择要选择最小的
    if samples_list is None: # 没有给定 samples_list, 就认为是全部
        samples_list = [cnt for cnt in range(self.samples_amount)]
    if available_division_feature_list is None: # 没有给定 available_division_feature_list,
认为全部可用
        available_division_feature_list = [cnt for cnt in range(self.features_number)]
    chosen_division_feature_id = 0
    min_gini_index = 1.0
    for division_feature_id in available_division_feature_list:
        temp_gini_index = self.computeOneFeatureGiniIndex(division_feature_id, samples_list)
        # print(temp_gini_index)
        if temp_gini_index < min_gini_index:
            chosen_division_feature_id = division_feature_id
            min_gini_index = temp_gini_index
    return chosen_division_feature_id, None
# 将三种选取最优划分特征的方法进行汇总 (不关心连续或非连续)
def getDivisionFeature(self, available_division_feature_list=None, samples_list=None,
get_division_feature_method=user.entropy_gain):
    if get_division_feature_method == user.entropy_gain:
        return self.selectDivisionFeatureByEntropyGain(available_division_feature_list,
samples_list)
    elif get_division_feature_method == user.gain_ratio:
        return self.selectDivisionFeatureByGainRatio(available_division_feature_list,
samples_list)
    elif get_division_feature_method == user.gini_index:
        return self.selectDivisionFeatureByGiniIndex(available_division_feature_list,
samples_list)
# 判断给定的 samples 再给定的 features 十分完全相同 (对是否连续都能处理)
def judgeSamplesHaveSameFeatures(self, features_list, samples_list):
    same_feature_flag = 1
    if len(samples_list) == 1:
        return 1
    for i in features_list: # 对特征循环
        first_sample_feature = self.features[samples_list[0]][i] # 给定的第 0 个样本所有特征的
值一定为新的
        for j in samples_list[1:]: # 对样本循环
            temp_feature = self.features[j][i]
            if temp_feature != first_sample_feature:
                same_feature_flag = 0
                return same_feature_flag
    return same_feature_flag

```

```

# 获取列表形式的标签值
def getLabelsPossibleValuesList(self):
    labels_possible_values_list = []
    for label_value in self.labels_possible_values:
        labels_possible_values_list.append(label_value)
    return labels_possible_values_list

# 拆分数据集和训练集
def seperateTrainSetAndTestSet(self, train_dataset_numbers=0):
    if train_dataset_numbers > self.samples_amount:
        quit(-1)
    else:
        self.train_sub_dataset_list = []
        self.test_sub_dataset_list = [i for i in range(self.samples_amount)] # 测试集挑剩下的

        while len(self.train_sub_dataset_list) < train_dataset_numbers:
            sample_idx = random.randrange(0, self.samples_amount) # 使用randrange, 上开下闭
            if sample_idx not in self.train_sub_dataset_list:
                self.train_sub_dataset_list.append(sample_idx)
        for sample_idx in self.train_sub_dataset_list:
            self.test_sub_dataset_list.remove(sample_idx)

        print("train_sub_dataset_list : ", self.train_sub_dataset_list)
        print("test_sub_dataset_list : ", self.test_sub_dataset_list)

if __name__ == "__main__":
    # melon_dataset = Dataset(dataset_path=user.DatasetPath, dataset_name=user.DatasetName)
    # melon_dataset.comprehensiveInitializeDataset(continuity_list=[1, 1])
    # melon_dataset.printDatasetInfo()
    #
    # # melon_dataset.computeOneFeatureEntropyGain(division_feature_id=0)
    # melon_dataset.selectDivisionFeatureByEntropyGain()
    pass

```

## (二) DecisionTreeNodeClass.py

```

# 1950083 自动化 刘智宇
import UserToolFunction

"""
节点类设计
类变量
1. 类型（是否为叶节点）
2. 子节点列表

```



3. 父节点

4. 当前节点划分特征

5. 子节点对应的划分特征

成员函数:

1. 打印

2. 向下遍历

"""

```
class DecisionTreeNode:
```

```
    global_node_id = 0 # 类共享变量 (在有新treenode 创建时会自增)
```

```
    # 定义类变量
```

```
    def __init__(self):
```

```
        self.node_id = DecisionTreeNode.global_node_id # 节点编号 (单个数值)
```

```
        DecisionTreeNode.global_node_id += 1 # 共享变量自增
```

```
        self.parentnode = None # 父节点 (单个节点)
```

```
        self.parentnode_id = None # 父节点编号 (单个数值)
```

```
        self.childnodes_list = [] # 子节点列表 (节点列表)
```

```
        self.childnodes_id_list = [] # 子节点编号列表 (数值列表)
```

```
        self.division_feature_id = None # 当前节点用于划分子节点的划分特征编号 (单个数值)
```

```
        self.childnode_division_feature_values = [] # 子节点对应划分特征的特征值 (特征值列表)
```

```
        # (对于连续值, 二分法, 存放相同的值, 默认左节点为'smaller', 右节点为'bigger')
```

```
        self.continuous_feature_split_value = None # 若当前的划分特征为连续值, 用于记录最佳划分特征
```

值

```
        self.include_samples = [] # 当前节点包含的样本编号 (数值列表)
```

```
        self.available_features_id = [] # 可用的划分特征列表, 从父节点的改成员变量删去父节点划分特征  
        所得 (数值列表)
```

```
        self.final_label = None # 对于叶节点, 给出归到这里的样本的最终决策结果 (标注的实际值)
```

```
        # print(self.global_node_id)
```

```
    # 输出节点信息 (节点都用编号表示)
```

```
    def getNodeInfo(self):
```

```
        UserToolFunction.printSeparateBar()
```

```
        print("node_id          : ", self.node_id)
```

```
        print("parentnode         : ", self.parentnode_id)
```

```
        print("childnodes_list      : ", self.childnodes_id_list)
```

```
        print("div_feature_id       : ", self.division_feature_id)
```

```
        print("continue_split_value : ", self.continuous_feature_split_value)
```

```
        print("include_samples      : ", self.include_samples)
```

```
        print("final_label          : ", self.final_label)
```

```
        UserToolFunction.printSeparateBar()
```

```
    # 给当前节点添加子节点, 同时更新当前节点及后续节点中的指向信息
```

```
    def attachChildNode(self, childnode, childnode_feature_value):
```

```
        # 父节点指向子节点
```

```
        self.childnodes_list.append(childnode)
```

```
        self.childnodes_id_list.append(childnode.node_id)
```

```
        # 子节点指向父节点
```

```

        childnode.parentnode = self
        childnode.parentnode_id = self.node_id
        # 父子节点其余信息调整
        self.childnode_division_feature_values.append(childnode_feature_value)
# 更新当前节点的划分特征
def updateDivisionProperty(self, div_feature):
    self.division_feature_id = div_feature
# 深度优先打印以当前节点为根节点的决策树
def printDecisionTreeDepthFirst(self):
    self.getNodeInfo()
    if len(self.childnodes_list): # 子节点列表非空
        for childnode in self.childnodes_list:
            childnode.printDecisionTreeDepthFirst()
# 广度优先打印以当前节点为根节点的决策树
def printDecisionTreeBreadthFirst(self):
    node_queue = [self]
    queue_pointer = 0
    while queue_pointer < len(node_queue):
        node_queue.extend(node_queue[queue_pointer].childnodes_list)
        queue_pointer += 1
    for node in node_queue:
        node.getNodeInfo()
def setIncludeSample(self, include_sample_list):
    self.include_samples = include_sample_list
def setAvailableFeaturesId(self, available_feature_id_list):
    self.available_features_id = available_feature_id_list
def saveDecisionTreeToFile(self, filename="DecisionTree.txt"):
    pass
def readDecisionTreeFromFile(self, filename="DecisionTree.txt"):
    pass
# 判断是不是叶子节点
def isLeaf(self):
    if not len(self.childnodes_id_list): # 子节点非空
        return True
    else: # 没有子节点
        return False
# 获取样本通过决策树得到的结果（递归方式）（传入样本的全部特征的值）
def getSampleLabelByDecisionTree(self, sample_features_list, features_continuity_list):
    if not self.isLeaf(): # 当前节点不为叶节点，需要进行细分
        if features_continuity_list[self.division_feature_id] == 1: # 当前节点的划分特征为连续特征
            if sample_features_list[self.division_feature_id] <=
self.continuous_feature_split_value: # 特征值不大于划分值
                use_childnode = self.childnodes_list[0] # 选择使用左节点（规定左节点为 smaller）

```

```

        else:
            use_childnode = self.childnodes_list[1] # 选择使用右节点（规定右节点为bigger）
            final_label = use_childnode.getSampleLabelByDecisionTree(sample_features_list,
features_continuity_list)
        else: # 当前节点的划分特征为离散特征
            sample_division_feature_value =
sample_features_list[self.division_feature_id] # 获取该样本对应当前节点的划分特征的值
            # 获取样本划分特征的值所对应的子节点划分特征的值的列表中的序号，即后续递归使用的子节点
            在当前节点的子节点列表中的序号
            division_feature_value_idx =
self.childnode_division_feature_values.index(sample_division_feature_value)
            use_childnode = self.childnodes_list[division_feature_value_idx]
            final_label = use_childnode.getSampleLabelByDecisionTree(sample_features_list,
features_continuity_list)
            # use_sub_node = self.childnodes_list[]
        return final_label
    else: # 当前节点为叶节点，当前节点的final_label 即为样本的分类最终结果
        return self.final_label
# 获取当前决策树混淆矩阵（字典形式）（对于是否连续不关心）
def getConfusionMatrixDict(self, dataset, test_list=None):
    # 先创建字典的字典，外层字典用于表示实际Label，内层字典用于表示通过决策树得到的Label
    confusion_matrix = {}
    for outer_label_value in dataset.labels_possible_values:
        confusion_matrix[outer_label_value] = {}
        for inner_label_value in dataset.labels_possible_values:
            confusion_matrix[outer_label_value][inner_label_value] = 0
    # 对每一个样本进行划分，按照实际Label 和决策树给出的Label 放入相应的混淆矩阵中
    if test_list is None: # 如果没有传入，则使用根节点的全部数据
        use_samples = self.include_samples
    else: # 传入测试样本集编号列表，则使用（对于iris 等拆分了的数据集）
        use_samples = test_list
    for sample_id in use_samples:
        outer_label_value = dataset.labels[sample_id] # 实际Label
        # 决策树给出的Label（传入一个样本的全部特征）
        inner_label_value = self.getSampleLabelByDecisionTree(dataset.features[sample_id],
dataset.features_continuity)
        confusion_matrix[outer_label_value][inner_label_value] += 1
    return confusion_matrix
if __name__ == "__main__":
    pass

```

### (三) DecisionTreeVisualize.py

```
# 1950083 自动化 刘智宇

# Thirdparty
import graphviz
import matplotlib.pyplot as plt
import numpy as np
import os.path

# User
from DecisionTreeNodeClass import DecisionTreeNode
from DatasetClass import Dataset

# 定义常量
leaf_shape = 'ellipse'
leaf_color = 'lightblue'
middle_shape = 'box'
middle_color = 'lightgreen'
node_fontsize = '20'
node_fontname = 'NSimSun'
node_style = 'filled'
edge_fontsize = '15'
edge_fontname = 'NSimSun'

# 设置
def setGraphEdgeAttribute(graph):
    graph.attr('edge', fontname=edge_fontname, fontsize=edge_fontsize)
def setGraphNodeAttribute(graph, node_type):
    if node_type == 'leaf': # 是叶节点
        graph.attr('node', shape=leaf_shape, style=node_style, color=leaf_color,
                    fontname=node_fontname, fontsize=node_fontsize)
    elif node_type == 'middle': # 不是叶节点
        graph.attr('node', shape=middle_shape, style=node_style, color=middle_color,
                    fontname=node_fontname, fontsize=node_fontsize)

# 只画当前节点
def drawOneTreeNode(graph, current_node, features_name_list, features_continuity):
    if current_node.isLeaf(): # 是叶节点
        setGraphNodeAttribute(graph, 'leaf') # 设置叶节点属性
        graph.node(name=str(current_node.node_id), label=str(current_node.final_label))
    else: # 不是叶节点
        setGraphNodeAttribute(graph, 'middle') # 设置中间节点属性
        if features_continuity[current_node.division_feature_id] == 1: # 连续特征
            graph.node(name=str(current_node.node_id),
label=str(features_name_list[current_node.division_feature_id]) + ' </> ' + "%.4f" %
current_node.continuous_feature_split_value)
        else: # 离散特征
```

```

        graph.node(name=str(current_node.node_id),
label=str(features_name_list[current_node.division_feature_id]) + ' = ?')
        return
# 画所有子节点和连线
def drawSubNodesAndEdges(graph, current_node, features_name_list, features_continuity):
    for childnode_idx in range(len(current_node.childnodes_id_list)): # childnode_idx 是指
childnode 在
        childnode = current_node.childnodes_list[childnode_idx]
        # 画子节点
        drawOneTreeNode(graph, childnode, features_name_list, features_continuity)
        # graph.view()
        # 画连线
        graph.edge(str(current_node.node_id), str(childnode.node_id),
                    label=str(current_node.childnode_division_feature_values[childnode_idx]))
        # print(current_node.childnode_division_feature_values[childnode_idx])
# 画整颗树 (bfs 方式)
def drawDecisionTree(root_node, feature_name_list, features_continuity, filename=""):
    # 创建 graph
    tree_graph = graphviz.Graph(name="DecisionTree", filename=filename+'DecisionTree',
                                directory="SavedDecisionTree", format='png')
    drawOneTreeNode(tree_graph, root_node, feature_name_list, features_continuity)
    setGraphEdgeAttribute(tree_graph)
    # tree_graph.view(cleanup=True)
    node_queue = [root_node] # 将根节点添加进节点列表
    queue_pointer = 0
    while queue_pointer < len(node_queue):
        node_queue.extend(node_queue[queue_pointer].childnodes_list)
        queue_pointer += 1
    for node in node_queue:
        drawSubNodesAndEdges(tree_graph, node, feature_name_list, features_continuity)
        # node.getNodeInfo()
    return tree_graph
# 将字典形式的混淆矩阵转为 np.array 形式
def dictConfusionMatrixToNpArray(confusion_mat_dict, show_flag=False):
    confusion_mat_list = []
    for i in confusion_mat_dict:
        temp_list = list(confusion_mat_dict[i].values())
        confusion_mat_list.append(temp_list)
        if show_flag is True:
            print(temp_list)
    return np.array(confusion_mat_list)
# 用 matplotlib 绘制混淆矩阵并保存
def drawConfusionMatrix(confusion_matrix_ndarray, label_values, fig_name):
    plt.figure()

```

```

plt.title('ConfusionMatrix')
# plt.axis('off')
plt.ylabel("Actual Label") # 纵轴
plt.xlabel("Decision Tree Generate Label") # 横轴
plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签
# 在坐标轴上显示feature_values
tick_marks = np.arange(len(label_values))
plt.xticks(tick_marks, label_values) # , rotation=45
plt.yticks(tick_marks, label_values)
# 显示混淆矩阵
plt.imshow(confusion_matrix_narray, cmap=plt.cm.Blues)
for i in range(len(label_values)):
    for j in range(len(label_values)):
        # 这里i, j 需要反过来, 否则会转置关系
        plt.text(j, i, confusion_matrix_narray[i][j], fontdict={'fontsize': 20},
ha='center', va='center')
    save_fig_path = os.path.join('SavedDecisionTree', fig_name+'ConfusionMatrix.png')
    plt.savefig(save_fig_path)
plt.show(block=False) # 一定要先保存图片再进行显示, 否则保存的图片为空白
plt.pause(3) # 显示3秒自动关闭窗口, 再plt.show中需要添加block=False
plt.close()
if __name__ == "__main__":
    conf_np_array = dictConfusionMatrixToNpArray({1: {1: 41, 2: 12, 3: 13}, 2: {3: 23, 4: 24, 5:
25}, 3: {5: 35, 6: 36, 7: 37}})
    # print(conf_np_array)
    drawConfusionMatrix(conf_np_array, ['da', 'sha', 'bi'], "haha")

```

#### (四) GenerateDecisionTree.py

```

# 1950083 自动化 刘智宇

# Thirdparty
# User
import UserToolFunction as user
from DecisionTreeNodeClass import DecisionTreeNode
from DatasetClass import Dataset
import DecisionTreeVisualize as viz
def generateDecisionTree(dataset, current_node, get_division_feature_method):
    labels_possible_values_dict =
dataset.countLabelsPossibleValuesByList(current_node.include_samples)
    most_possible_label_value = user.getMaxValueIndex(labels_possible_values_dict) # 得到当前序

```



列最大的值，对列表和数组都适用

```
# 判断样本是否全属于同一类别，如果是则将该treenode 标记为该类的叶节点
if len(labels_possible_values_dict) == 1:
    current_node.final_label = list(labels_possible_values_dict.keys())[0] # 将此节点作为叶子节点，并最终标记其输出Label
    return

# 判断特征集是否为空或所有样本在特征集上取值相同，则将该节点标记为叶节点，其Label 为最可能的
if (not len(current_node.available_features_id)) or \
    dataset.judgeSamplesHaveSameFeatures(current_node.available_features_id,
current_node.include_samples):
    current_node.final_label = most_possible_label_value
    # print(most_possible_label_value)
    return

# 得到最优划分特征编号，以及最优划分值（连续特征独有）
best_division_feature_id, best_division_feature_value =
dataset.getDivisionFeature(current_node.available_features_id,
                                current_node.include_samples, get_division_feature_method)

best_division_feature_name = dataset.features_name[best_division_feature_id]
print("best_division_feature_name: ", best_division_feature_name)
sub_node_available_features = current_node.available_features_id.copy()

if not dataset.features_continuity[best_division_feature_id]: # 对于非连续特征，作为划分属性后，其子节点不应再使用
    sub_node_available_features.remove(best_division_feature_id) # 选出了最佳划分特征，后面的决策应不受该特征影响

current_node.division_feature_id = best_division_feature_id
# print(current_node.division_feature_id)

# 根据最优划分特征，将样本集进行拆分
split_dict = dataset.splitSamplesList(division_feature_id=best_division_feature_id,
samples_list=current_node.include_samples,
                                continuity_split_value=best_division_feature_value)
# print(split_dict)

if dataset.features_continuity[best_division_feature_id]: # 对于连续特征，只有bigger、smaller两个节点
    current_node.continuous_feature_split_value = best_division_feature_value # 设置连续特征的最佳划分值

    # 设置smaller 节点
    smaller_sub_node = DecisionTreeNode() # 添加smaller 子节点
    current_node.attachChildNode(smaller_sub_node, 'smaller')
    smaller_sub_node.available_features_id = sub_node_available_features
    smaller_sub_node.setIncludeSample(split_dict['smaller'])
    generateDecisionTree(dataset, smaller_sub_node, get_division_feature_method)

    # 设置bigger 节点
    bigger_sub_node = DecisionTreeNode() # 添加bigger 子节点
```

```

        current_node.attachChildNode(bigger_sub_node, 'bigger')
        bigger_sub_node.available_features_id = sub_node.available_features
        bigger_sub_node.setIncludeSample(split_dict['bigger'])
        generateDecisionTree(dataset, bigger_sub_node, get_division_feature_method)
    else: # 对于离散特征, 需要将划分特征的每一个特征值都建立树节点
        for feature_value in dataset.features_possible_values[best_division_feature_id]: # 需要
给该特征的每一个可能值都建立树节点, 即使没有可能值
            sub_node = DecisionTreeNode() # 生成新的子节点
            current_node.attachChildNode(sub_node, feature_value) # 将子节点与当前节点进行连接
            sub_node.available_features_id = sub_node.available_features
            if split_dict.get(feature_value, 0): # 如果拆分出的字典键值中有 feature_value
                sub_node.setIncludeSample(split_dict[feature_value])
                generateDecisionTree(dataset, sub_node, get_division_feature_method) # 递归调用
            else: # 如果拆分出的字典键值中没有 feature_value, 即没有出现过该种情况
                sub_node.final_label = most_possible_label_value # 作为叶子节点, final_label 设为
最可能的值
        return
if __name__ == "__main__":
    # 西瓜数据集
    # 创建数据集
    dataset = Dataset(dataset_path=user.DatasetPath, dataset_name=user.DatasetName)
    dataset.comprehensiveInitializeDataset(continuity_list=user.DatasetContinuityList)
    dataset.printDatasetInfo()
    dataset.seperateTrainSetAndTestSet()
    # 创建决策树
    user.printSeparateBar()
    print("Start generating a decision tree.")
    root_node = DecisionTreeNode() # 创建根节点
    root_node.setIncludeSample([cnt for cnt in range(dataset.samples_amount)]) # 根节点包含所有
样本
    root_node.setAvailableFeaturesId([cnt for cnt in range(dataset.features_number)]) # 根节点
拥有所有划分特征的选择权
    # 生成决策树
    generateDecisionTree(dataset=dataset, current_node=root_node,
get_division_feature_method=user.get_division_feature_method)
    print("Finish generating a decision tree")
    user.printSeparateBar()
    # # 打印决策树信息广度优先方式方式, 方便横向查看
    # root_node.printDecisionTreeBreadthFirst()
    # 打印决策树信息深度优先方式方式, 方便横向查看
    # # root_node.printDecisionTreeDepthFirst()
    # 使用Graphviz 库图形化展示决策树, 并将图片进行保存
    tree_graph = viz.drawDecisionTree(root_node, dataset.features_name,
dataset.features_continuity, user.viz_decision_tree_name)

```

```

tree_graph.view(cleanup=True) # 清除了png 图片以外的其他多余文件

confusion_matrix_dict = root_node.getConfusionMatrixDict(dataset) # 获得字典形式的混淆矩阵
confusion_matrix_narray = viz.dictConfusionMatrixToNpArray(confusion_matrix_dict) # 字典形
式的混淆矩阵转为np.array 形式

viz.drawConfusionMatrix(confusion_matrix_narray, dataset.getLabelsPossibleValuesList(),
fig_name=user.viz_decision_tree_name)

# # IRIS 数据集或其他需要进行拆分 test、train 的数据集
# # 创建数据集
# dataset = Dataset(dataset_path=user.DatasetPath, dataset_name=user.DatasetName)
# dataset.comprehensiveInitializeDataset(continuity_list=user.DatasetContinuityList)
# dataset.printDatasetInfo()
# dataset.seperateTrainSetAndTestSet(user.TrainSamplesAmount) # 拆分测试集、训练集（流出法）
#
# # 创建决策树
# user.printSeparateBar()
# print("Start generating a decision tree.")
# root_node = DecisionTreeNode() # 创建根节点
# root_node.setIncludeSample(dataset.train_sub_dataset_list) # 根节点包含训练样本
# root_node.setAvailableFeaturesId([cnt for cnt in range(dataset.features_number)]) # 根节
点拥有所有划分特征的选择权
# generateDecisionTree(dataset=dataset, current_node=root_node,
get_division_feature_method=user.get_division_feature_method) # 生成决策树
# print("Finish generating a decision tree")
# user.printSeparateBar()
#
# # # 打印决策树信息广度优先方式方式，方便横向查看
# root_node.printDecisionTreeBreadthFirst()
# # # 打印决策树信息深度优先方式方式，方便横向查看
# # # root_node.printDecisionTreeDepthFirst()
#
# confusion_matrix_dict = root_node.getConfusionMatrixDict(dataset,
dataset.test_sub_dataset_list) # 获得字典形式的混淆矩阵
# confusion_matrix_narray = viz.dictConfusionMatrixToNpArray(confusion_matrix_dict) # 字典
形式的混淆矩阵转为np.array 形式
# viz.drawConfusionMatrix(confusion_matrix_narray, dataset.getLabelsPossibleValuesList(),
fig_name=user.viz_decision_tree_name)
#
# # 使用 Graphviz 库图形化展示决策树，并将图片进行保存
# tree_graph = viz.drawDecisionTree(root_node, dataset.features_name,
dataset.features_continuity, user.viz_decision_tree_name)
# tree_graph.view(cleanup=True) # 清除了png 图片以外的其他多余文件

```

## (五) UserToolFunction.py

```
# 1950083 自动化 刘智宇

# Thirdparty
import os.path
import math

# 对于不同数据集情况需要适当修改（例如数据集特征是否有名称，数据集是否有编号）
dataset_header = 0 # 目前只支持数据集特征有名称，后续可添加
dataset_index_col = None # 默认是没有index_col
# 常量定义
DatasetsFolder = "Datasets"
# DatasetName = "MelonDataset2"
# DatasetPath = os.path.join(DatasetsFolder, "MelonDataset2.csv")
# DatasetContinuityList = None
# DatasetName = "MelonDataset2alpha"
# DatasetPath = os.path.join(DatasetsFolder, "MelonDataset2alpha.csv")
# DatasetContinuityList = None
DatasetName = "MelonDataset3"
DatasetPath = os.path.join(DatasetsFolder, "MelonDataset3.csv")
DatasetContinuityList = [0, 0, 0, 0, 0, 0, 1, 1]
# DatasetName = "MelonDataset3alpha"
# DatasetPath = os.path.join(DatasetsFolder, "MelonDataset3alpha.csv")
# DatasetContinuityList = [1, 1]
# DatasetName = "IrisDataset"
# DatasetPath = os.path.join(DatasetsFolder, "IrisDataset.csv")
# DatasetContinuityList = [1, 1, 1, 1]
# TrainSamplesAmount = 70
# 全局变量定义
entropy_gain = 0
gain_ratio = 1
gini_index = 2
get_division_feature_method = entropy_gain # 注意对于连续值，目前只支持entropy_gain
get_division_feature_methods_name = ["ByEntropyGain", "ByGainRatio", "GainIndex"]
viz_decision_tree_name = DatasetName +
get_division_feature_methods_name[get_division_feature_method]
def printSeparateBar(length=40, sign="*"):
    print(sign*length)
def getMaxValueIndex(sequence):
    if type(sequence) == dict:
        key_list = list(sequence.keys())
        value_list = list(sequence.values())
        max_value = max(value_list)
```

```
        max_value_index = key_list[value_list.index(max_value)]
        return max_value_index
    elif type(sequence) == list:
        max_value = max(sequence)
        max_value_index = sequence.index(max_value)
        return max_value_index
    else:
        print("Can't handle this sequence.")
        return None
def pLog2p(possibility):
    possibility = float(possibility)
    return possibility * math.log(possibility, 2)
if __name__ == "__main__":
    pass
```