决策树 decision tree

1. 简要介绍

决策树，英文为 decision tree。决策树是一种**分类**学习方法，基于**树结构**进行决策。

决策树的基本原理

1. 一般的，一棵决策树包含一个**根结点**、若干个**内部结点**和若干个**叶结点**；其中叶结点对应决策结果，其他每个结点对应一个属性测试；根结点包含样本全集，其他每个节点包含的样本集合根据属性测试的结果被划分到子结点中。
2. 决策树学习基本算法如下，

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　属性集 $A = \{a_1, a_2, \ldots, a_d\}$.
过程： 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 　将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 　将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 　为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: 　**if** D_v 为空 **then**
12: 　　将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: 　**else**
14: 　　以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: 　**end if**
16: **end for**
输出： 以 node 为根结点的一棵决策树

决策树的生成是一个**递归过程**，三种情形会导致递归返回，

- **当前结点包含的样本全属于同一类别**，无需划分
- **当前结点属性集为空 or 所有样本在所有属性上取值相同**，无法划分 ==> 将当前结点标记为叶结点，并将其类别设定为该结点所含样本最多的类别 (利用当前节点的后验分布)
- **当前结点包含的样本集合为空**，不能划分 ==> 将当前结点标记为叶结点，并且将其类别设定为其父结点所包含样本最多的类别 (将父结点的样本分布作为当前结点的先验分布)

属性划分方法

决策树学习的关键是如何选择**最优的划分属性**. 一般而言，随着划分过程不断进行，我们希望决策树的**分支结点所包含的样本尽可能属于同一类别**，即结点的"纯度" (purity) 越来越高. 关于选择最优的划分属性，主要有三种方法：信息增益、增益率基尼系数。

I信息增益（ID3 决策树，Iterative Dichotomiser）

1. “**信息熵**” (information entropy) 是**度量样本集合纯度**最常用的一种指标，当前**样本集合 D 的信息熵**定义如下 (假定当前样本集合 D 中第 k 类样本所占的比例为 $p_k(k=1,2,\ldots,|\gamma|)$)

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

对于信息熵有以下结论

- **Ent(D) 的值越小，则 D 的纯度越高**
- 约定：若 $p=0$, 则 $plog_2 p=0$
- $min\ Ent(D)=0$，$max\ Ent(D)=log_2|\gamma|$

2. “**信息增益**” (information gain) 可以用于**选择最优划分属性，属性 a 对样本集合 D 进行划分所获得的信息增益**定义如下(假定离散属性 a 有 V 个可能的取值 $\{a^1,a^2,\ldots,a^V\}$，若使用属性 a 对样本集合 D进行划分，则会产生 V 个分支结点，其中第 v 个分支结点包含了 D 中所有在属性 a 上取值为 a^v 的样本，记为 D^v)，

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

其中，由于考虑到不同分支结点所包含的样本数不同，因此给分支结点赋予权重 $|D^v|/|D|$ ，即样本数越多的分支结点的影响越大。对于信息增益有以下结论

- **Gain(D,a) 越大，则表示使用属性 a 来进行划分所获得的 “纯度提升” 越大**

3. 信息增益准则的特点：信息增益准则**对可取值数目较多的属性有所偏好**
4. 最优划分属性的选择：从候选划分属性中**选择信息增益最大的属性**

$$a_*=\underset{a\in A}{arg\,max}\ Gain(D,a)$$

II 增益率（C4.5 决策树）

1. “**增益率**” (gain ration) 的定义如下，

$$Gain\_ratio(D,a) = \frac{Gain(D,a)}{IV(a)} \tag{3}$$

$$IV(a) = -\sum_{v=1}^{V} \frac{|D^v|}{|D|} log_2 \frac{|D^v|}{|D|} \tag{4}$$

其中，IV(a) 称之为属性 a 的固有值 (intriinsic value)，一般而言，属性 a 的可能取值数目越多 (V 越大)，则固有之 IV(a) 的值通常越大

2. 增益率准则的特点：增益率准则**对可取值数目较少的属性有所偏好**
3. 最优划分属性的选择：启发式
   - 先从候选划分属性中**找到信息增益高于平均水平的属性**
   - 再从中选择**增益率最高的**

III 基尼系数（CART 决策树，Classification and Regression BinaryTree）

1. “基尼值” (Gini) 可以用来**度量数据集的纯度**，**数据集 D 的基尼值**定义如下，

$$\begin{aligned} Gini(D) &= \sum_{k=1}^{|\gamma|} \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{|\gamma|} p_k^2 \end{aligned} \tag{5}$$

其中，Gini(D) 表示从数据集 D 中随机抽取两个样本，其类别标记不一致的概率。对于基尼值，有以下结论，

- **Gini(D) 越小，则数据集 D 的纯度越高**

2. “基尼指数” (Gini index) 用于**选择最优划分属性**，属性 a 的基尼指数定义为，

$$Gini\_index(D,a) = \sum_{v=1}^{V} \frac{|D^v|}{D} Gini(D^v) \tag{6}$$

3. 最优划分属性的选择：从候选划分属性中**选择基尼指数最小的属性**

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

连续值处理

由于连续属性的可取值数目不再有限，因此不能直接根据其可取值对结点进行划分，于是可以采取**连续属性离散化**技术对连续属性进行处理。C4.5 决策树算法中采用的**二分法** (bi-partition) 对连续属性进行处理，其步骤如下，

给定样本集合 D 和连续属性 a，假定 a 在 D 上出现了 n 个不同的取值，将这些值从小到大进行排序，记为 $\{a^1, a^2, \dots, a^n\}$.

1. 求得**候选划分点集合 T**：基于 T 中的划分点 t 可以将 D 分为子集 D_t^- 和 D_t^+，其中 D_t^- = {在属性a上取值 $\leq t$的样本}，D_t^+ = {在属性a上取值 $> t$的样本}。对于连续属性 a，可以求得包含 n-1 个元素的候选划分点集合，

$$T_a = \{\frac{a^i + a^{i+1}}{2} | 1 \leq i \leq n-1\}$$

即将区间 $[a^i, a^{i+1})$ 的中位点 $\frac{a^i+a^{i+1}}{2}$ 作为候选划分点.

2. 基于信息增益**选取最优的划分点**：Gain(D,a,t) 是样本集合基于划分点 t 二分后的信息增益，最优划分点即使 Gain(D,a,t) 最大化的划分点

$$\begin{aligned} Gain(D,a) &= \max_{t \in T_a} Gain(D,a,t) \\ &= \max_{t \in T_a} Ent(D) - \sum_{\lambda \in \{-,+\}} \frac{|D_t^\lambda|}{|D|} Ent(D_t^\lambda) \end{aligned}$$

注意：**若当前结点划分属性为连续属性，则该属性还可以作为其后代的划分属性**

2. 模型训练步骤

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_m, y_m)\}$;
属性集 $A = \{a_1, a_2, \dots, a_d\}$.
过程： 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: **if** D_v 为空 **then**
12: 将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: **else**
14: 以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: **end if**
16: **end for**
输出： 以 node 为根结点的一棵决策树

3. 数据集介绍

DATASET: Loan-Approval-Prediction-Dataset, **4296 × 13 features** (9 integer, 2 string, 1 id, 1 other)

The loan approval dataset is a collection of financial records and associated information used to **determine the eligibility of individuals or organizations for obtaining loans** from a lending institution. It includes various factors such as **cibil score, income, employment status, loan term, loan amount, assets value, and loan status**.

4. 模型训练代码

DecisionTree.py

> 该 python 文件定义了一个 DecisionTree 的类，通过实例化类对象和调用相关方法实现**决策树模型**的训练和测试。
>
> 1. 实例化决策树模型 model=DecisionTree()
>
>   - 可以指定模型的相关参数：min_samples_split 最小可分样本数量；max_depth 结点最大深度；criterion 度量最优划分属性的参考标准
>
> 2. 模型训练(实例化模型后) model.fit(train)
>
>   - 参数解释：train 是训练数据集(包含标签)
>   - 该方法默认使用信息增益准则选择最优化分属性，递归生成决策树
>
> 3. 模型测试(模型训练后) model.accuracy(test)
>
>   - 参数解释：test 是测试数据集(包含标签)
>   - 该方法用于对模型进行测试，返回模型预测的精确度(预测准确样本数量/测试样本数量)

```
import numpy as np
import pandas as pd

from LoanApprovalPredict.model.DecisionTree.BinaryTree import *


class DecisionTree:
    """ 决策树模型，用于二分类 """

    def __init__(self, min_samples_split=5, max_depth=6, criterion="entropy"):
        self.min_samples_split = min_samples_split  # 结点可分最小样本量(小于则直接划分为叶结点)
        self.max_depth = max_depth  # 决策树的最大深度(根节点是零层)
        self.criterion = criterion  # 最优化分属性的判断依据
        # 一般而言，有信息增益 entropy, 增益率 entropy_ratio, 基尼系数 gini 三种方法，本决策树类仅实现了第一种方法
        self.tree = None  # 创建一个空二叉树作为决策树

    # 训练决策树
```

```
    def fit(self, data):
        if not self.tree:  # 决策树为空
            self.tree = self.create_tree(data)

    # 递归生成决策树
    def create_tree(self, data, depth=0):
        newNode = TreeNode()  # 新建结点
        data_label = data.iloc[:, -1]  # 数据集标签列

        # 1.递归返回判断
        # 当前结点包含的样本全部属于同一类别，则无需划分
        if len(data_label.value_counts()) == 1:
            newNode.node_class = data_label.values[0]  # 结点类别
            newNode.node_type = NodeStatus.LEAF  # 结点类型(叶结点)
            newNode.samples = data.shape[0]  # 结点对应的样本数量
            newNode.depth = depth
            return BinaryTree(newNode)
        # 当前结点属性集为空(只剩数据标签列) or 所有样本在所有属性上取值相同 or 当前结点的属性数量少于5
        if data.shape[1] == 1 or \
                all([len(data[fea].value_counts()) == 1 for fea in data.columns[:-1]]) or \
                data.shape[0] < 5 or \
                depth == self.max_depth:
            newNode.node_class = DecisionTree.get_most_label(data)
            newNode.node_type = NodeStatus.LEAF  # 结点类型(叶结点)
            newNode.samples = data.shape[0]  # 结点对应的样本数量
            newNode.depth = depth
            return BinaryTree(newNode)

        # 2.递归生成决策树
        best_feature_info = DecisionTree.get_best_feature(data)  # 最优化分属性相关信息(连续和离散属性返回的变量数目不同)
        if len(best_feature_info) == 3:  # 连续属性
            best_feature_type = FeatureStatus.CONTINUOUS
        else:  # 离散属性
            best_feature_type = FeatureStatus.DISCRETE

        if best_feature_type == FeatureStatus.CONTINUOUS:  # 对于连续属性
            best_feature, information_gain, threshold = best_feature_info
        else:  # 对于离散属性
            best_feature, information_gain = best_feature_info
            threshold = data[best_feature].value_counts().keys()[0]

        # 填充结点信息
        newNode.feature_name = best_feature
        newNode.feature_type = best_feature_type
```

```
        newNode.threshold = threshold
        newNode.entropy = information_gain
        newNode.samples = data.shape[0]
        newNode.node_type = NodeStatus.NON_LEAF
        newNode.node_class = DecisionTree.get_most_label(data)
        newNode.depth = depth

        # 生成树对象(待插入左右子树)
        newTree = BinaryTree(newNode)

        # 生成左右子树
        if best_feature_type == FeatureStatus.CONTINUOUS:  # 对于连续属性
            data_t_less = data.loc[data[best_feature] <= threshold].drop(best_feature, axis=1)
            data_t_greater = data.loc[data[best_feature] > threshold].drop(best_feature, axis=1)
            if not data_t_less.empty:  # 添加左子树
                child = self.create_tree(data_t_less, depth=depth + 1)
                newTree.insert_left(child)
            if not data_t_greater.empty:  # 添加右子树
                child = self.create_tree(data_t_greater, depth=depth + 1)
                newTree.insert_right(child)
        else:  # 对于离散属性
            data_equals = data.loc[data[best_feature] == threshold].drop(best_feature, axis=1)
            data_unequals = data.loc[data[best_feature] != threshold].drop(best_feature, axis=1)
            if not data_equals.empty:  # 添加左子树
                child = self.create_tree(data_equals, depth=depth + 1)
                newTree.insert_left(child)
            if not data_unequals.empty:  # 添加右子树
                child = self.create_tree(data_unequals, depth=depth + 1)
                newTree.insert_right(child)

        # 返回生成的树对象
        return newTree

    # 返回模型预测数据集的准确度
    def accuracy(self, data):
        data_label = data.iloc[:, -1]
        data_predict = self.predict(data)

        count = 0
        data_scale = len(data_label)

        for i in range(data_scale):
            if data_label.iloc[i] == data_predict.iloc[i]:
                count += 1

        acc = count / data_scale
```

```
        return acc

    # 预测数据分类
    def predict(self, data):
        samples = data.iloc[:, :-1]  # 将数据集去除标签列
        result = []  # 存储预测结果

        for i in range(data.shape[0]):
            sample = samples.iloc[i, :]

            current_tree = self.tree  # 从根结点进行判断
            current_node_type = current_tree.node.node_type  # 当前结点的状态(是不是叶子节点)
            current_feature_type = current_tree.node.feature_type  # 当前属性的状态(离散还是连续)

            while current_node_type == NodeStatus.NON_LEAF:  # 只要没有遍历到叶结点就一直遍历
                if current_feature_type == FeatureStatus.DISCRETE:  # 离散属性
                    if current_tree.node.threshold == sample[current_tree.node.feature_name]:  # 遍历到左子树
                        current_tree = current_tree.left_child
                    else:  # 遍历到右子树
                        current_tree = current_tree.right_child
                else:  # 连续属性
                    if current_tree.node.threshold >= sample[current_tree.node.feature_name]:
                        current_tree = current_tree.left_child
                    else:
                        current_tree = current_tree.right_child

                current_node_type = current_tree.node.node_type
                current_feature_type = current_tree.node.feature_type

            result.append(current_tree.node.node_class)

        return pd.Series(result)

    # 计算数据集的信息熵
    @staticmethod
    def cal_information_entropy(data):
        data_label = data.iloc[:, -1]  # 数据集标签列
        label_class = data_label.value_counts()  # 标签频次统计
        Ent = 0  # 数据集的信息熵
        for k in label_class.keys():
            p_k = label_class[k] / len(data_label)
            Ent += -p_k * np.log2(p_k)
        return Ent

    # 计算数据集以属性 a 划分的信息增益
    # 如果是离散属性，返回信息增益; 如果是连续属性，返回信息增益，最优化分点
```

```
    @staticmethod
    def cal_information_gain(data, a):
        Ent = DecisionTree.cal_information_entropy(data)  # 数据集信息熵
        feature_class = data[a].value_counts()  # 属性为 a 的数据列进行频次统计

        # 判断属性 a 是连续还是离散
        if len(feature_class.keys()) < 3:  # 离散(属性为 a 的数据列只有两种取值，或更少)
            feature_type = FeatureStatus.DISCRETE
        else:
            feature_type = FeatureStatus.CONTINUOUS

        # 根据属性类型求解信息增益
        if feature_type == FeatureStatus.DISCRETE:  # 离散属性
            gain = 0
            for k in feature_class.keys():
                weight = feature_class[k] / data.shape[0]
                data_v = data.loc[data[a] == k]  # data 在属性 a 上取值为 v 的样本集 (.loc 根据 data[a]==v 的真值 Series
对象，选择符合条件的样本)
                Ent_v = DecisionTree.cal_information_entropy(data_v)
                gain += weight * Ent_v
            return [Ent - gain]
        else:  # 连续属性
            # 将属性为 a 的数据列取出、升序排序、去重、重置索引
            sorted_feature = data[a].sort_values().drop_duplicates().reset_index(drop=True)
            # 1. 获取候选划分点集合
            T = []  # 候选划分点集合
            for i in range(len(sorted_feature) - 1):
                t = (sorted_feature[i] + sorted_feature[i + 1]) / 2
                T.append(t)
            # 2. 基于信息增益准则选取最优的划分点
            # 获取不同候选划分点下的样本集合的信息增益
            T_gain = []  # 不同候选划分点下的样本集合的信息增益
            for t in T:
                # 以候选划分点 t 划分数据集 data
                data_t_less = data.loc[data[a] <= t]
                data_t_greater = data.loc[data[a] > t]
                # 计算划分后的两个数据集的信息熵
                Ent_t_less = DecisionTree.cal_information_entropy(data_t_less)
                Ent_t_greater = DecisionTree.cal_information_entropy(data_t_greater)
                # 计算划分后的两个数据集的权重
                weight_t_less = np.sum(data[a] <= t) / data.shape[0]
                weight_t_greater = np.sum(data[a] > t) / data.shape[0]
                # 计算信息增益
                gain_a_t = Ent - weight_t_less * Ent_t_less - weight_t_greater * Ent_t_greater
                T_gain.append(gain_a_t)
            # 选择最优候选划分点
```

```
            best_t_gain = max(T_gain)  # 最大信息增益
            threshold = T[T_gain.index(best_t_gain)]  # 最优划分点
            return [best_t_gain, threshold]

    # 返回数据集中大多数样本所属的类
    @staticmethod
    def get_most_label(data):
        data_label = data.iloc[:, -1]
        label_sort = data_label.value_counts(sort=True)
        most_label = label_sort.keys()[0]
        return most_label

    # 获取最优划分属性
    # 如果是离散属性，返回最优化分属性，对应信息增益；如果是连续属性，返回最优化分属性，对应信息增益，对应划分点
    @staticmethod
    def get_best_feature(data):
        features = data.columns[0:-1]  # 所有属性名
        res = {}  # 字典存储信息增益，key 是属性名，value 是对应的信息增益
        for a in features:
            temp = DecisionTree.cal_information_gain(data, a)[0]  # 计算不同属性的信息增益
            res[a] = temp
        res = res.items()  # 以列表的形式返回可遍历的元组数组，每个元组由一个键值对组成(属性:增益)
        res = sorted(res, key=lambda x: x[1], reverse=True)  # 基于信息增益进行降序排序

        best_feature = res[0][0]  # 最优划分属性

        # 判断 best_feature 是连续还是离散
        feature_class = data[best_feature].value_counts()  # 属性为 a 的数据列进行频次统计
        if len(feature_class.keys()) < 3:  # 离散(属性为 a 的数据列只有两种取值，或更少)
            feature_type = FeatureStatus.DISCRETE
        else:
            feature_type = FeatureStatus.CONTINUOUS

        if feature_type == FeatureStatus.DISCRETE:  # 离散属性
            best_information_gain = res[0][1]
            return [best_feature, best_information_gain]
        else:  # 连续属性
            best_information_gain, threshold = DecisionTree.cal_information_gain(data, best_feature)
            return [best_feature, best_information_gain, threshold]
```

BinaryTree.py

> 此 python 文件定义若干类，介绍如下，

1. NodeStatus: 结点状态枚举类，区分结点是叶结点还是非叶节点
2. FeatureStatus: 结点最优化分属性状态枚举类，区分该属性是连续属性还是离散属性
3. TreeNode: 结点类型

- 属性
  - feature_name 当前结点对应的数据集的最优化分属性
  - feature_type 最优化分属性的类型(离散or连续)
  - threshold 最优化分属性的判定分界值
  - entropy 最优化分属性的信息增益
  - samples 当前结点对应的数据集的样本数量
  - node_type 当前结点的类型(叶结点or非叶节点)
  - node_class 当前结点对应的数据集的大多数样本所属的类别
- 方法
  - print_info() 当前结点的重要信息
  - print() 打印当前结点的重要信息

4. BinaryTree: 二叉树类型

- 属性
- tree 当前二叉树的数据(包括结点、左子树、右子树)
- node 当前二叉树的结点
- left_child 当前二叉树的左子树
- right_child 当前二叉树的右子树
- 方法
  - insert_left() 插入左子树
  - insert_right() 插入右子树
  - level_order_traversal() 非递归层次遍历二叉树，并返回遍历信息
  - print_level_order_traversal() 打印二叉树的层次遍历结果
  - visualize() 可视化当前二叉树

```
from enum import Enum
from queue import Queue
from graphviz import Digraph
import time


class NodeStatus(Enum):
    """ 结点类型 """
    LEAF = 0
    NON_LEAF = 1
```

```
class FeatureStatus(Enum):
    """ 属性类型 """
    DEFAULT = 2
    CONTINUOUS = 1
    DISCRETE = 0


class TreeNode:
    """ 结点类型 """

    # 创建一个结点实例
    def __init__(self, feature_name=None, feature_type=None, threshold=None, entropy=None, samples=None,
node_type=None,
                 node_class=None):
        self.feature_name = feature_name  # 结点(最优化分)属性, 无则 None, 以下结点属性都意为结点最优属性
        self.feature_type = feature_type  # 结点属性的类型(FeatureStatus.CONTINUOUS/FeatureStatus.DISCRETE),
无则 None
        self.threshold = threshold  # 结点属性判定阈值, 无则 None
        """
            对于离散类型属性(FeatureStatus.CONTINUOUS), 意味着如果 data[feature_name] == threshold, 去左子树,
否则去右子树
            对于连续类型属性(FeatureStatus.DISCRETE), 意味着如果 data[feature_name] <= threshold, 去左子树, 否
则去右子树
        """
        self.entropy = entropy  # 结点属性的信息增益, 无则 None
        self.samples = samples  # 结点包含的样本数据数量, 无则 None
        self.node_type = node_type  # 结点类型 (NodeStatus.LEAF/NodeStatus.NON_LEAF), 必须有值
        self.node_class = node_class  # 结点类别 (即该结点包含的全部样本，大多数属于什么类别，或者说样本标
签是什么), 必须有值
        self.depth = 0  # 结点深度

    # 获取结点信息
    def print_info(self):
        node_class = "Approval" if self.node_class == 1 else "Rejected"

        if self.node_type == NodeStatus.LEAF:
            node_info = "node info:\n" \
                        f"class={node_class}"
        elif self.node_type == NodeStatus.NON_LEAF:
            distinguish_flag = "==" if self.feature_type == FeatureStatus.DISCRETE else "<="
            node_info = "node info:\n" \
                        f"\t{self.feature_name}{distinguish_flag}{self.threshold}\n" \
                        f"\tentropy={self.entropy}\n" \
                        f"\tsamples={self.samples}\n" \
                        f"\tclass={node_class}\n"
        else:
```

```
            raise Exception("nodes whose type is not known cannot be printed")

        return node_info

    # 打印结点信息
    def print(self):
        node_info = self.print_info()
        print(node_info)


class BinaryTree:
    """ 二叉树类型: 列表表示 """

    """
        一个二叉树由一个列表表示: 列表第一个元素是树的结点, 列表第二个元素是树的左子树, 列表第三个元素是树的右子树
    """

    # 创建一个二叉树实例
    def __init__(self, node):
        self.tree = [node, None, None]
        self.node = self.tree[0]  # TreeNode 类型
        self.left_child = self.tree[1]  # BinaryTree 类型
        self.right_child = self.tree[2]  # BinaryTree 类型

    # 插入左子树
    def insert_left(self, child):
        if not self.left_child:  # 当前树的左子树为空，则将新的子树作为左子树插入
            self.tree[1] = child
            self.left_child = self.tree[1]
        else:  # 如果当前树的左子树不为空，则报错，表示子树插入位置错误
            raise Exception("the left child of the current tree is not empty, cannot be inserted")

    # 插入右子树
    def insert_right(self, child):
        if not self.right_child:  # 当前树的左子树为空，则将新的子树(也有可能是结点)作为左子树插入
            self.tree[2] = child
            self.right_child = self.tree[2]
        else:  # 如果当前树的左子树不为空，则报错，表示子树插入位置错误
            raise Exception("the right child of the current tree is not empty, cannot be inserted")

    # 非递归层次遍历二叉树并返回遍历列表信息，队列实现
    def level_order_traversal(self):
        queue = Queue()
        queue.put(self)  # 存放树对象(根据树对象的 node 属性可以判断这树是不是叶结点)
        traversal_res = []  # 存放前序遍历结果
```

```
    while not queue.empty():
        current_tree = queue.get()  # 取出一个树对象
        traversal_res.append(current_tree.node)
        if current_tree.left_child:
            queue.put(current_tree.left_child)
        if current_tree.right_child:
            queue.put(current_tree.right_child)

    return traversal_res

# 打印二叉树的层次遍历结果
def print_level_order_traversal(self):
    traversal_res = self.level_order_traversal()
    for node in traversal_res:
        node.print()

# 可视化二叉树，通过队列进行非递归的实现
def visualize(self, directory='test_output', file_name="VisualizedBinaryTree", file_format="png"):
    # 创建有向图，设置相关参数
    graph = Digraph(name=file_name, filename=file_name,
                    directory=directory, format=file_format)  # 创建一个有向图，并且设置文件保存地址及输出格式
    graph.graph_attr['dpi'] = '300'  # 设置输出图片的分辨率
    graph.attr('node', shape='box')  # 统一修改 node 对象的形状为 box

    # 为每一个结点分配一个唯一的 ID
    id_counter = 0

    # 使用字典存储结点和对应的 ID (KEY:结点;VALUE:ID), 从而可以根据结点查询对应的 ID
    node_to_id = {}

    # 创建根结点
    root_node = self.node
    root_id = f"node_{id_counter}"
    node_to_id[root_node] = root_id

    graph.node(root_id, label=root_node.print_info())

    # 根据二叉树的内容创建 node 对象和 edge 对象
    queue = Queue()
    queue.put(self)
    while not queue.empty():
        current_tree = queue.get()  # 取出一个树对象
        parent_id = node_to_id[current_tree.node]
        # 如果这个树对象有子树，则创建两个子节点，并且和父结点相连；否则从队列取出下一个树对象
        # 若这个树对象只有一个子树，则另一个子树生成一个不可见的 node 对象
```

```
            # 生成子树 node 对象后，将子树添加进队列，留用下一层的遍历
            if not current_tree.left_child and not current_tree.right_child:  # 当前树对象没有子树
                continue

            # 处理左子树
            if current_tree.left_child:  # 当前树对象有左子树
                id_counter += 1
                left_id = f"node_{id_counter}"
                node_to_id[current_tree.left_child.node] = left_id
                graph.node(left_id, label=current_tree.left_child.node.print_info())  # 创建左子树结点
                graph.edge(parent_id, left_id)  # 连接父结点和左子树结点
                queue.put(current_tree.left_child)
            else:  # 当前树对象没有左子树，则随机生成一个不可见的 node 对象并连接
                id_counter += 1
                left_id = f"node_{id_counter}"
                graph.node(left_id, label="", shape="point")  # 生成一个具有随机值的结点(表示空结点)
                graph.edge(parent_id, left_id, style='invis')  # 将父结点和空结点连接，并且不显示

            # 处理右子树
            if current_tree.right_child:  # 当前树对象有右子树
                id_counter += 1
                right_id = f"node_{id_counter}"
                node_to_id[current_tree.right_child.node] = right_id
                graph.node(right_id, label=current_tree.right_child.node.print_info())
                graph.edge(parent_id, right_id)
                queue.put(current_tree.right_child)
            else:  # 当前树对象没有右子树，则随机生成一个不可见的 node 对象并连接
                id_counter += 1
                right_id = f"node_{id_counter}"
                graph.node(right_id, label="", shape="point")  # 生成一个具有随机值的结点(表示空结点)
                graph.edge(parent_id, right_id, style='invis')  # 将父结点和空结点连接，并且不显示
        graph.render(view=True)
```

4. 模型测试代码

loan_approval_DT.py

> 此 python 文件调用了 Util 文件中的若干方法，对决策树模型 DecisionTree进行：
>
> 1. **测试集上的模型性能测试**
> 2. **可视化**决策树
> 3. 绘制**学习曲线**，观察训练集规模对模型拟合程度的影响

```
from Utils import *

# 导入数据集
# 划分数据集
train, test = loan_approval_data_processed(directory='../data/loan_approval_dataset.csv', minmax=False, scale=0.75)

# 训练模型
dt_model = DecisionTree()
dt_model.fit(train)

# 决策树可视化
dt_model.tree.visualize()

# 模型在测试集上的精度
accuracy = dt_model.accuracy(test)
print("模型的准确率为:%2.2f" % (accuracy * 100), "%")  # 模型的准确率为: 97.38 %

# 绘制学习曲线 learning curve
# 横轴：训练集的样本数量，纵轴：模型在训练集和测试集上的准确度
processed_data = loan_approval_data_processed(directory='../data/loan_approval_dataset.csv', minmax=False,
split=False)
draw_learning_curve(processed_data, DecisionTree(), start=0.01, end=1.0, step=0.05)
```

Utils.py

```
"""
   工具文件，内含一些工具函数
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from sklearn import preprocessing
from scipy.interpolate import make_interp_spline
from LoanApprovalPredict.model.LogisticRegression.LogisticRegression_newton import *
from LoanApprovalPredict.model.DecisionTree.DecisionTree import *

""" loan_approval_data: 4296 rows × 13 columns, 且数据集干净，无需缺失值处理等
    loan_id                 int64  贷款批准样本的 id (无用特征)
    no_of_dependents          int64  feature1: Number of Dependents of the Applicant(家属个数), int
    education               object  feature2: 受教育情况(Graduate/Not Graduate), str; Mapping(Graduate:1, Not
Graduate:0)
    self_employed           object  feature3: 是否为自雇人士(Yes/No), str; Mapping(Yes:1, No:0)
    income_annum              int64  feature4: 年收入, int
```

```
    loan_amount               int64  feature5: 贷款数额, int
    loan_term                 int64  feature6: 贷款期限, int
    cibil_score               int64  feature7: 信用分数, int
    residential_assets_value     int64  feature8: 住宅资产价值, int
    commercial_assets_value      int64  feature9: 商业资产价值, int
    luxury_assets_value          int64  feature10: 奢侈品资产价值, int
    bank_asset_value          int64  feature11: 银行资产价值, int
    loan_status               object  y: 是否借贷(Approval/Rejected), str; Mapping(Approval:1, Rejected:0)
    综上所述，一个样本总共有 11 个属性 or 特征，1 个值
"""

# 导入、预处理、[划分]数据
def loan_approval_data_processed(directory='../data/loan_approval_dataset.csv', minmax=True, split=True,
scale=0.75):
    """
    :param directory: loan_approval_predict 数据集位置
    :param minmax: 是否进行归一化
    :param split: 是否将数据集划分为训练集和测试集
    :param scale: 训练集比例
    :return: train, test
    """
    # 1. 导入数据
    loan_approval_data = pd.read_csv(directory)

    # 2. 数据预处理(由于提供的数据比较干净：无缺失值，因此只需要进行①删除指定列②将字符串映射为
值，这两项工作即可)
    loan_approval_data.drop("loan_id", axis=1, inplace=True)  # 删除 loan_id 这一列的数据
    loan_approval_data.columns = loan_approval_data.columns.str.strip()  # 清除列名的前后空格
    # loan_approval_data.columns = [name.strip() for name in loan_approval_data.columns]  # 清除列名的前后空格
    # 2.1 将属性 education 中的 Graduated 映射为 1，Not Graduated 映射为 0
    loan_approval_data.loc[:, "education"] = loan_approval_data.loc[:,
                                             "education"].str.strip()  # 清除列 "education" 每一项的前后空格
    loan_approval_data.loc[:, "education"] = loan_approval_data.loc[:, "education"].map(
        {'Graduate': 1, 'Not Graduate': 0}).astype(int)
    loan_approval_data["education"] = pd.to_numeric(loan_approval_data["education"],
                                                    errors='coerce')  # 更改该列数据的属性为 int 类型
    # 2.2 将属性 self_employed 中的 Yes 映射为 1，No 映射为 0
    loan_approval_data.loc[:, "self_employed"] = loan_approval_data.loc[:,
                                                 "self_employed"].str.strip()  # 清除列 "self_employed" 每一项的前后空格
    loan_approval_data.loc[:, "self_employed"] = loan_approval_data.loc[:, "self_employed"].map(
        {'Yes': 1, 'No': 0}).astype(int)
    loan_approval_data["self_employed"] = pd.to_numeric(loan_approval_data["self_employed"],
                                                        errors='coerce')  # 更改该列数据的属性为
    # 2.3 将属性 loan_status 中的 Approval 映射为 1，Rejected 映射为 0
    loan_approval_data.loc[:, "loan_status"] = loan_approval_data.loc[:,
                                               "loan_status"].str.strip()  # 清除列 "loan_status" 每一项的前后空格
```

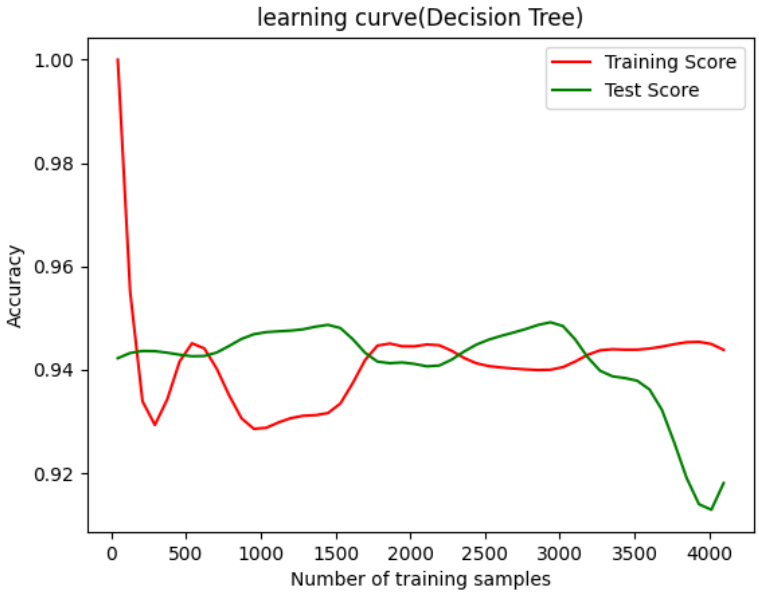
```
    loan_approval_data.loc[:, "loan_status"] = loan_approval_data.loc[:, "loan_status"].map(
        {'Approved': 1, 'Rejected': 0}).astype(int)
    loan_approval_data["loan_status"] = pd.to_numeric(loan_approval_data["loan_status"], errors='coerce')  # 更改该列数据的属性为
    # 2.4 对数据进行归一化操作
    processed_data = loan_approval_data
    if minmax:
        minmax_scaler = preprocessing.MinMaxScaler()
        minmax_data = minmax_scaler.fit_transform(loan_approval_data)
        processed_data = pd.DataFrame(minmax_data, columns=loan_approval_data.columns)

    # 3. 划分训练集和验证集
    if split:
        train = processed_data.sample(frac=scale, random_state=int(time.time()))
        test = processed_data.drop(train.index)
        train.reset_index(drop=True, inplace=True)
        test.reset_index(drop=True, inplace=True)

        return train, test
    else:
        return processed_data


# 将数据集划分为样本数据、标签
def dataset_split(data):
    X = data.iloc[:, 0:-1]
    y = data.iloc[:, -1]
    return X, y


def draw_learning_curve(data, model, start=0.001, end=0.3, step=0.001):
    plt.xlabel("Number of training samples")  # 横轴
    plt.ylabel("Accuracy")  # 纵轴

    # 计算出对应不同训练集规模的时，模型在训练集和测试集合上的比重
    train_size_list = list(np.arange(start, end, step))  # 训练集占全部数据集的比重
    number_of_training_samples_list = [int(size * data.shape[0]) for size in
                                       train_size_list]  # 训练集的样本数量(横坐标的值)
    train_accuracy_list = []  # 训练集随样本数的准确度取值(纵坐标)
    test_accuracy_list = []  # 测试集随样本数的准确度取值(纵坐标)
    # 获取两个图像的纵坐标取值
    for frac in train_size_list:
        # a = time.time()
        # 按照预定的比例划分数据集和测试集合
        train = data.sample(frac=frac, random_state=int(time.time()))
        test = data.drop(train.index)
```

```
    if isinstance(model, LogisticRegression_newton):
        plt.title("learning curve(Logistic Regression)")  # 标题
        # 重置索引
        train.reset_index(drop=True, inplace=True)
        test.reset_index(drop=True, inplace=True)
        # 划分训练集和测试集的样本数据、标签
        X_train, y_train = dataset_split(train)
        X_test, y_test = dataset_split(test)
        # 模型训练
        model.fit(X_train, y_train)
        # 添加纵坐标值
        train_accuracy = model.accuracy(X_train, y_train)
        train_accuracy_list.append(train_accuracy)
        test_accuracy = model.accuracy(X_test, y_test)
        test_accuracy_list.append(test_accuracy)
    else:
        plt.title("learning curve(Decision Tree)")  # 标题
        # 重置索引
        train.reset_index(drop=True, inplace=True)
        test.reset_index(drop=True, inplace=True)
        # 模型训练
        model.fit(train)
        # 添加纵坐标值
        train_accuracy = model.accuracy(train)
        train_accuracy_list.append(train_accuracy)
        test_accuracy = model.accuracy(test)
        test_accuracy_list.append(test_accuracy)

    # b = time.time()
    # print("\nfrac=%.2f" % frac)
    # print("time=%.2f" % (b - a))
    # print("train_acc%.2f" % train_accuracy)
    # print("test_acc%.2f" % test_accuracy)

# 绘图
# 对 x、y_train 插值
number_of_training_samples_list_smooth = np.linspace(min(number_of_training_samples_list),
                                                     max(number_of_training_samples_list), 50)
train_accuracy_list_smooth = make_interp_spline(number_of_training_samples_list, train_accuracy_list)(
    number_of_training_samples_list_smooth)
plt.plot(number_of_training_samples_list_smooth, train_accuracy_list_smooth, color='r', label='Training Score')
# 对 y_test 插值
test_accuracy_list_smooth = make_interp_spline(number_of_training_samples_list, test_accuracy_list)(
    number_of_training_samples_list_smooth)
plt.plot(number_of_training_samples_list_smooth, test_accuracy_list_smooth, color='g', label='Test Score')
```

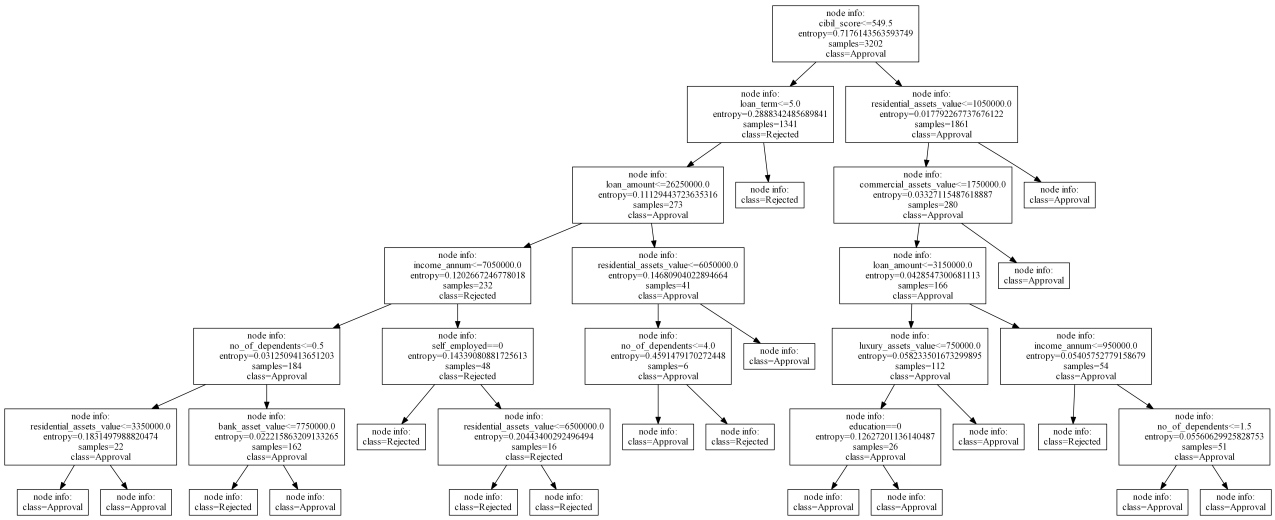
```
    plt.legend(loc='best')
    plt.show()


# 返回模型训练时间
def time_consumption(data, model):
    start = time.time()
    if isinstance(model, DecisionTree):
        model.fit(data)
    elif isinstance(model, LogisticRegression_newton):
        X, y = data
        model.fit(X, y)
    else:
        raise Exception
    end = time.time()
    return end - start
```

6. 结果分析

1. 给定训练集比例 0.75，测试集比例 0.25，模型的准确率为 96.63 %
2. 模型的**学习曲线**绘制如下，发现
   - 当训练集规模 < 300时，模型出现**欠拟合**状态，即模型在训练集上表现良好，在测试集上表现不佳
   - 当训练集规模 > 300 时，模型表现**趋于稳定**，预测准确率在 92% 左右



3. 据 graphviz 对决策树可视化如下


node info:
cibil_score<=549.5
entropy=0.7176143563593749
samples=3202
class=Approval
node info:
loan_term<=5.0
entropy=0.2888342485689841
samples=1341
class=Rejected
node info:
residential_assets_value<=1050000.0
entropy=0.017792267737676122
samples=1861
class=Approval
node info:
loan_amount<=26250000.0
entropy=0.11129443723635316
samples=273
class=Approval
node info:
class=Rejected
node info:
commercial_assets_value<=1750000.0
entropy=0.03327115487618887
samples=280
class=Approval
node info:
class=Approval
node info:
income_annum<=7050000.0
entropy=0.12026672467778018
samples=232
class=Rejected
node info:
residential_assets_value<=6050000.0
entropy=0.14680904022894664
samples=41
class=Approval
node info:
loan_amount<=3150000.0
entropy=0.04285473006811113
samples=166
class=Approval
node info:
class=Approval
node info:
no_of_dependents<=0.5
entropy=0.031250941365120 3
samples=184
class=Approval
node info:
self_employed==0
entropy=0.14339080881725613
samples=48
class=Rejected
node info:
no_of_dependents<=4.0
entropy=0.4591479170272448
samples=6
class=Approval
node info:
class=Approval
node info:
luxury_assets_value<=750000.0
entropy=0.05823350167329 9895
samples=112
class=Approval
node info:
income_annum<=950000.0
entropy=0.05405752779158679
samples=54
class=Approval
node info:
residential_assets_value<=3350000.0
entropy=0.1831497988820474
samples=22
class=Approval
node info:
bank_asset_value<=7750000.0
entropy=0.022215863209133265
samples=162
class=Approval
node info:
class=Rejected
node info:
residential_assets_value<=6500000.0
entropy=0.20443400292496494
samples=16
class=Rejected
node info:
class=Approval
node info:
class=Rejected
node info:
education==0
entropy=0.12627201136140487
samples=26
class=Approval
node info:
class=Approval
node info:
class=Rejected
node info:
no_of_dependents<=1.5
entropy=0.05560629925828753
samples=51
class=Approval
node info:
class=Approval
node info:
class=Approval
node info:
class=Rejected
node info:
class=Approval
node info:
class=Rejected
node info:
class=Rejected
node info:
class=Approval
node info:
class=Approval
node info:
class=Approval
node info:
class=Approval
node info:
class=Approval
node info:
class=Approval