

机器学习 第三次作业

决策树作业

201250076 袁家乐

2023 年 3 月 18 日

一、数据的分析与处理

本次给出的数据集为药品相关，格式为.csv，以训练数据集为例对数据集的各属性值进行分析见下表：

属性名	分析	示例
recordId	此为记录的 id 值，不具备分类意义，应清洗掉	95202
drugName	此为药品名称，同样不具备分类意义，应清洗掉	Drysol
condition	此为药品的使用情况，非常重要，比 sideEffects 更精细，适宜作分类的属性	Abnormal Uterine Bleeding
reviewComment	此为用户评价，为自然语言文本，不适宜分类，且其核心特征已被包括进 condition 和 sideEffects，故应清洗掉	"I only had 2 SHOTS!! Been bleeding non stop for 6 months straight! I'm off the shot now for over 11 days and still bleeding, have a gyn appt. Scared what could it be???"
date	记录产生的日期，不具备分类意义，应清洗掉	7-Mar-16
usefulCount	药品有用的计数，适宜用作为分类，但此为连续值属性，应做离散化	28

sideEffects	药品副作用程度，适宜用作分类，共 5 种取值	No Side Effects
rating	药品等级类别，1~5 五等	3

结合上述的分析，在数据预处理的部分，我只保留了训练集中的 condition, usefulCount, sideEffects 和 rating 这四列，其余属性列均清洗干净。特别地，对于 condition 这一列，有一些取值为“7 users found this comment helpful.”，开头的数字可变换，对于这些取值，我将其归并为两个值：若开头的数字为 0，则归并为“comment useless”；否则，归并为“comment useful”。部分数据的 condition 列为空，结合实际意义，为其赋予“no condition”的取值，而不是将其作为缺失值在构造 C4.5 决策树时处理。以此实现了数据的去噪和清晰工作。

```
def data_pre_process():
    """
    加载、预处理（清洗、去掉无关列）三份csv文件
    """
    # 读入training.csv, 清洗掉无关列, 对condition列做一定的清洗
    # 只保留condition,usefulCount和sideEffects
    with open('training.csv', 'r', encoding='utf-8') as train_csv:
        reader = csv.DictReader(train_csv)
        for row in reader:
            clean_line = ['', '', '', '']
            clean_line[0] = row.get('condition')
            if clean_line[0].find('users found this comment helpful.') != -1:
                if clean_line[0][0:1] == '0':
                    clean_line[0] = 'comment helpless'
                else:
                    clean_line[0] = 'comment helpful'
            if clean_line[0] == '':
                clean_line[0] = 'no condition'
            clean_line[1] = row.get('usefulCount')
            clean_line[2] = row.get('sideEffects')
            clean_line[3] = row.get('rating')
            train_data_set.append(clean_line)
    print('training.csv loaded: SUCCESS!')
```

二、决策树的设计原理和核心代码

（一）设计原理

考虑到需要对 usefulCount 这一连续值属性进行离散化，我最终选取了 C4.5 算法来构建决策树。

整体设计的核心原理如下：

A、在构建每一个决策树结点时，依次考虑以下几个步骤：

- 1、若当前样本均属于同一个类别，则将此类别标记为叶节点并返回，叶节点以元组形式存放分类名、分类下样本数、非此分类下样本数。
- 2、若当前样本的属性列表已然为空，则选取样本中出现次数最多的类别作

为叶节点并返回，叶节点以元组形式存放分类名、分类下样本数、非此分类下样本数。

3、选择最佳的分类属性

1) 针对连续属性需要先离散化，计算出其最佳的划分值，以此划分值将连续属性划分为两个取值。

2) 计算各属性的信息增益和信息增益率（需要根据属性取值划分后的样本信息熵计算、对 IV 的计算等）

3) 以“在信息增益超过平均水平的候选属性中信息增益率最高的”为标准选定最佳的分类属性

4、若最佳分类属性是离散值，则针对该分类属性的每一个取值构建子树（构建子树的输入样本集中需要清洗掉对应的属性列）

5、若最佳分类属性是连续值，则以“<”作为分类条件，依次递归地构建左右子树（由于连续值属性可以作为其后代结点的划分属性，故此处无需对子树的输入样本集进行清洗）

B、在使用构建出的 C4.5 决策树做分类时，依次考虑以下步骤：

1、首先获取当前的根结点，判断用作分类的属性是否为连续值

2、若为离散值，则依次遍历各子结点进行匹配，匹配到的子结点若非叶结点，则递归查询；否则，将叶结点的结果保存。

3、若为连续值，则依次遍历左右子结点进行匹配，匹配到的子结点若非叶结点，则递归查询；否则，将叶结点结果保存。

说明：对于叶结点的结果保存，若该叶结点存放的非此分类下样本数不为 0，则将其数量平均分配增加到其它各分类下。

4、最终权值最高的分类即为结果分类。

（二）核心代码解说

1、构建 C4.5 分类树核心方法

在当前样本均属于同一类别（完美分类），或当前属性列表为空（无可奈何，只能聊取样本数最多的分类），或无法选出最优的分类属性（无可奈何，只能聊取样本数最多的分类）这三种情形下构造叶结点并返回。

```
def create_my_tree(dataset, labels, is_discrete):
    """
    构建C4.5决策树
    :param dataset: 数据集
    :param labels: 属性列表
    :param is_discrete: 属性是否是离散值列表
    :return: tree C4.5决策树
    """
    # 取训练集的最后一列形成类别列表
    class_list = [data[-1] for data in dataset]

    # 如果当前样本均属于同一个类别，则将此类别标记为叶节点并返回
    if class_list.count(class_list[0]) == len(class_list):
        return (class_list[0], len(dataset), 0)

    # 如果属性列表已然为空，则返回出现次数最多的类别
    if len(dataset[0]) == 1:
        majority_class = Counter(class_list).most_common()
        return (majority_class[0][0], majority_class[0][1], len(dataset) - majority_class[0][1])

    # 选择最佳的分类属性
    best_index, best_split_value = choose_best_label(dataset, is_discrete)
    if best_index == -1:
        # 若无最佳的分类属性，则返回出现次数最多的类别
        majority_class = Counter(class_list).most_common()
        return (majority_class[0][0], majority_class[0][1], len(dataset) - majority_class[0][1])
```

对于最佳分类属性为离散值的，对所有取值递归地建立子树，注意需在构造子树的样本集中去掉已经用过的当前属性列。

```
# 最佳分类属性是离散值属性的情形
if is_discrete[best_index] or best_split_value == -999.0:
    label_name = labels[best_index]
    my_tree = {label_name: {}}
    # 针对此属性完成分类后，将此属性从列表中去除掉
    new_labels = copy.copy(labels)
    new_is_discrete = copy.copy(is_discrete)
    del (new_labels[best_index])
    del (new_is_discrete[best_index])
    # 对该属性的所有取值建立子树
    label_values = [data[best_index] for data in dataset]
    unique_values = set(label_values)
    for value in unique_values:
        sub_labels = new_labels[:]
        sub_is_discrete = new_is_discrete[:]
        sub_dataset = []
        for data in dataset:
            if data[best_index] == value:
                # 新构造的子数据集需要去掉已完成分类的属性列
                tmp_data_line = data[:best_index]
                tmp_data_line.extend(data[best_index + 1:])
                sub_dataset.append(tmp_data_line)
        my_tree[label_name][value] = create_my_tree(sub_dataset, sub_labels, sub_is_discrete)
```

对于最佳分类属性为连续值的，用小于号构建分类条件，递归地构建左右子树，连续值属性可用作后代结点的划分属性，无需清洗掉。

```

# 最佳分类属性是连续值属性情形
else:
    label_name = labels[best_index] + '<' + str(best_split_value)
    my_tree = {label_name: {}}
    # 连续值属性不应当被去除
    new_labels = labels[:]
    new_is_discrete = is_discrete[:]
    # 分别构建左右子树
    value_left = 'Y'
    value_right = 'N'
    sub_dataset_left = []
    sub_dataset_right = []
    for data in dataset:
        if float(data[best_index]) < best_split_value:
            sub_dataset_left.append(data)
        elif float(data[best_index]) > best_split_value:
            sub_dataset_right.append(data)
    my_tree[label_name][value_left] = create_my_tree(sub_dataset_left, new_labels, new_is_discrete)
    my_tree[label_name][value_right] = create_my_tree(sub_dataset_right, new_labels, new_is_discrete)

return my_tree

```

2、选择最佳的分类属性核心方法

由于 C4.5 算法要求以“在信息增益超过平均水平的候选属性中信息增益率最高的”，故此处需计算和存储信息增益、信息增益率的结果，由于连续值属性在确定最佳划分值的时候也需反复计算信息增益，为提升效率、避免冗余操作，故在处理连续属性时，将确定最佳划分值、计算信息增益和信息增益率融合进一个方法中实现。对于仅有一个取值的属性，显然其不应具备“最佳划分值”，信息增益和信息增益率也均应当为 0.0。

```

# 选择最佳的分类属性
def choose_best_label(dataset, is_discrete):
    """
    选择最佳的分类属性
    先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择信息增益率最高的
    :param dataset: 数据集
    :param is_discrete: 属性是否为离散值
    :return int 最佳划分的属性下标, float 针对连续值属性的最佳划分值
    """
    # 最佳划分的属性下标
    best_index = -1
    # 最佳划分的信息增益率
    best_gain_ratio = 0.0
    # 针对连续值属性的最佳划分值
    best_split_value = -999.0
    # 计算信息增益以获取平均信息增益，同时计算信息增益率，针对连续属性值划分其
    avg_gain = 0.0
    split_value_list = [-999.0 for k in range(len(is_discrete))]
    gain_list = [0.0 for k in range(len(is_discrete))]
    gain_ratio_list = [0.0 for k in range(len(is_discrete))]

```

```

for i in range(0, len(is_discrete), 1):
    # 处理离散值属性
    if is_discrete[i]:
        info_gain, info_gain_ratio = calculate_gain_ratio_discrete(dataset, i)
        avg_gain += info_gain
        gain_list[i] = info_gain
        gain_ratio_list[i] = info_gain_ratio
    # 处理连续值属性
    else:
        split_value, info_gain, info_gain_ratio = calculate_best_split_value(dataset, i)
        if split_value != -999.0:
            avg_gain += info_gain
            split_value_list[i] = split_value
            gain_list[i] = info_gain
            gain_ratio_list[i] = info_gain_ratio
avg_gain = avg_gain / len(is_discrete)

```

在计算出平均信息增益和各属性的信息增益率后，即可按照“在信息增益超过平均水平的候选属性中信息增益率最高的”确定最佳的分类属性，此外还应当返回连续值属性的最佳划分值（若无或为离散值属性，此值默认为-999.0）。

```
# 筛选出最佳的分类属性
for i in range(0, len(is_discrete), 1):
    if gain_ratio_list[i] > best_gain_ratio and gain_list[i] > avg_gain:
        best_index = i
        best_split_value = split_value_list[i]
return best_index, best_split_value
```

3、计算连续值属性的最佳划分值，同时计算出信息增益和信息增益率方法

为了避免冗余计算，在确定最佳划分值的同时，计算出信息增益和信息增益率。对于最佳划分值的确定，首先将连续值属性取值去重和排序，再依次算出各个相邻属性取值的平均数，作为候选的划分值，计算其信息增益，选择信息增益最大的候选划分值作为最优的划分值。

```
# 计算连续值属性的最佳划分值，同时计算出信息增益和信息增益率
def calculate_best_split_value(dataset, index):
    """
    计算连续值属性的最佳划分值，同时计算出信息增益和信息增益率
    :param dataset: 数据集
    :param index: 属性下标
    :return: float 最佳划分值, float 最佳划分值下计算出的信息增益, float 最佳划分值下计算出的信息增益率
    """
    # 对指定的属性列做排序和去重
    label_list = [int(data[index]) for data in dataset]
    sorted_unique_labels = sorted(list(set(label_list)))
    for i in range(0, len(sorted_unique_labels), 1):
        sorted_unique_labels[i] = str(sorted_unique_labels[i])
    # 若该属性列只有一个取值，则无需计算最佳划分值
    if len(sorted_unique_labels) == 1:
        return -999.0, 0.0, 0.0
    # 计算最佳划分值及其信息增益、信息增益率
    ans_gain = calculate_entropy(dataset)
    iv = 0.0
    best_split_value = -1.0
    min_num = inf
    dataset_len = len(dataset)
```

```
for i in range(0, len(sorted_unique_labels) - 1, 1):
    split_value = (float(sorted_unique_labels[i]) + float(sorted_unique_labels[i + 1])) / 2
    # 以此划分值划分出左右两边的数据集
    left_dataset = []
    right_dataset = []
    for data in dataset:
        if float(data[index]) < split_value:
            left_dataset.append(data)
        elif float(data[index]) > split_value:
            right_dataset.append(data)
    # 由于整体的信息熵是一致的，此处仅需要做数据间的比较，故只需计算信息熵的加权和即可
    left_weight = float(len(left_dataset)) / float(dataset_len)
    right_weight = float(len(right_dataset)) / float(dataset_len)
    num = left_weight * calculate_entropy(left_dataset) + right_weight * calculate_entropy(right_dataset)
    if num < min_num:
        min_num = num
        best_split_value = split_value
        iv = -1 * (left_weight * log(left_weight, 2) + right_weight * log(right_weight, 2))
    ans_gain = ans_gain - min_num
    if iv == 0.0:
        iv = 0.00000001
    ans_gain_ratio = ans_gain / iv
    return best_split_value, ans_gain, ans_gain_ratio
```


4、计算用离散值属性划分样本集获得的信息增益和信息增益率方法

对于离散值属性，首先对属性的取值做去重和排序，再针对各个属性取值计算该属性取值下的样本分类的带权信息熵与 IV 分量，最后将上述值各自做加和，进一步得到信息增益和 IV，计算出信息增益率。对于仅有一个取值的属性，其信息增益和信息增益率均为 0.0。

```
# 计算用离散值属性划分样本集获得的信息增益和信息增益率
def calculate_gain_ratio_discrete(dataset, index):
    """
    计算用离散值属性划分样本集获得的信息增益和信息增益率
    :param dataset: 数据集
    :param index: 属性下标
    :return: float 信息增益的数值, float 信息增益率的数值
    """
    ans_gain = calculate_entropy(dataset)
    iv = 0.0
    dataset_len = len(dataset)
    # 对指定的属性列做排序和去重
    label_list = [data[index] for data in dataset]
    sorted_unique_labels = sorted(list(set(label_list)))
    if len(sorted_unique_labels) == 1:
        return 0.0, 0.0

    for label in sorted_unique_labels:
        # 依据属性值划分数据集，计算各自的带权信息熵
        sub_dataset = []
        for data in dataset:
            if data[index] == label:
                sub_dataset.append(data)
        sub_weight = float(len(sub_dataset)) / float(dataset_len)
        ans_gain -= sub_weight * calculate_entropy(sub_dataset)
        iv -= sub_weight * log(sub_weight, 2)
    if iv == 0.0:
        iv = 0.00000001
    ans_gain_ratio = ans_gain / iv
    return ans_gain, ans_gain_ratio
```

相关的计算公式如下：

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in \text{values}(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f)$$

$$\text{Gain_ratio}(D, a) = \frac{\text{Gain}(D, a)}{IV(a)}$$

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

5、计算信息熵方法

针对给定的数据集，计算出其在分类下的数量，由此进一步计算出对应的信息熵。

```

# 计算信息熵
def calculate_entropy(dataset):
    """
    计算信息熵
    :param dataset: 数据集
    :return: float 信息熵的数值
    """
    dataset_len = len(dataset)
    # 计算各分类的样本数
    class_counts = {}
    for data in dataset:
        tmp_class = data[-1]
        if tmp_class not in class_counts.keys():
            class_counts[tmp_class] = 0
        class_counts[tmp_class] += 1
    ans = 0.0
    for key in class_counts:
        rate = float(class_counts[key]) / float(dataset_len)
        ans = ans - rate * log(rate, 2)
    return ans

```

相关的计算公式如下：

$$H(x) = - \sum_{i=1}^n p(x_i) \log_2 p(x_i)$$

6、主程序入口

整个程序的整体流程为“数据预处理和加载—构造 C4.5 分类树—验证集验证—对测试集进行分类—将测试结果写入文件”。其中验证部分使用了 `sk-learn` 库的 `f1_score`，采用 `micro` 和 `macro` 两种指标进行验证评价。对测试集的分类结果保存在 `testing_result.csv` 中。程序各流程部分执行完后，会在控制台输出完成信息，验证评价分数也打印在控制台上。

```

if __name__ == '__main__':
    # 数据预处理
    data_pre_process()
    # 构建分类树
    my_tree = create_my_tree(train_data_set, train_labels, labels_is_discrete)
    print('C4.5 tree built: SUCCESS!')

    # 使用验证集进行验证
    true_class_list = []
    predict_class_list = []
    for data in validation_data_set:
        # 分别使用Micro F1和Macro F1进行验证集评估
        micro_f1 = f1_score(y_true=true_class_list, y_pred=predict_class_list, average="micro")
        macro_f1 = f1_score(y_true=true_class_list, y_pred=predict_class_list, average="macro")

        print('Micro F1: {}'.format(micro_f1))
        print('Macro F1: {}'.format(macro_f1))

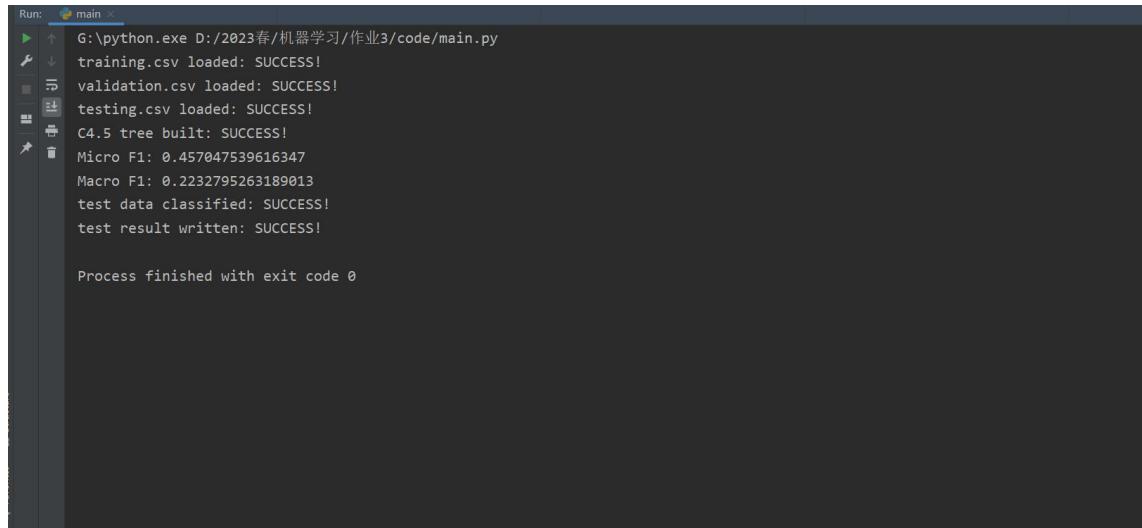
    # 对测试集进行分类
    for data in test_data_set:
        print('test data classified: SUCCESS!')

    # 测试结果写入文件中
    with open("testing_result.csv", "w", encoding="utf-8", newline="") as f:

```


三、验证集评估结果

控制台截图如下。可见 Micro F1 的分数为 0.457047539616347, Macro F1 的分数为 0.2232795263189013。整体的分类效果还算不错。此外,测试集的分类结果保存在了 testing_result.csv 中。



```
Run: main
G:\python.exe D:/2023春/机器学习/作业3/code/main.py
training.csv loaded: SUCCESS!
validation.csv loaded: SUCCESS!
testing.csv loaded: SUCCESS!
C4.5 tree built: SUCCESS!
Micro F1: 0.457047539616347
Macro F1: 0.2232795263189013
test data classified: SUCCESS!
test result written: SUCCESS!

Process finished with exit code 0
```