山东大学 计算机科学与技术 学院

模式识别与机器学习 课程实验报告

学号：201900170249	姓名：李阳	班级：智能 19
实验题目：决策树		
实验学时：4h	实验日期：May 7, 2021	

硬件环境

- Lenovo Legion Y7000P 2020H(16GB DDR4), Intel Core i7-10750H
- Windows 10, Chinese version

软件环境

- Visual Studio Code 1.55.2

实验目的

- 利用已有的数据集 wine data 实现决策树
- 利用 k=10 折交叉验证评估决策树
- 决策树可视化

实验记录

1 C4.5 决策树

C4.5 算法是用于生成决策树的一种经典算法，是 ID3 算法的一种延伸和优化。
C4.5 算法对 ID3 算法主要做了一下几点改进：

1. 通过信息增益率选择分裂属性，克服了 ID3 算法中通过信息增益倾向于选择拥有多个属性值的属性作为分裂属性的不足。
2. 能够处理离散型和连续型的属性类型，即将连续型的属性进行离散化处理。
3. 构造决策树之后进行剪枝操作。
4. 能够处理具有缺失属性值的训练数据。

通过读取 ex6Data.csv 的数据，可以发现 11 个特征均为连续型属性，且整体数据量庞大需要剪枝操作，因此本次实验选用 C4.5 决策树是十分合适的。

1.1 分裂属性的选择：信息增益率

分裂属性选择的评判标准是决策树算法之间的根本区别。区别于 ID3 算法通过信息增益选择分裂属性，C4.5 算法通过信息增益率选择分裂属性。
设 D 是类标记元组训练集，类标号属性具有 m 个不同值，m 个不同类 $C^i,\quad i=1,2,\cdots,m$，C_D^i 是 D 中 C^i 类的元组的集合，$|D|$ 和 $|C_D^i|$ 分别是 D 和 C_D^i 中的元组个数。
对于一个数据集合 D，其信息熵为

$$Info(D)=-\sum_{i=1}^{m}p_i\log_2(p_i)$$

现在假定按照属性 A 划分 D 中的元组，且属性 A 将 D 划分成 v 个不同的类。在该划分之后，新的信息熵为

$$Info_A(D)=\sum_{j=1}^{v}\frac{|D_j|}{|D|}\times Info(D_j)$$

由此得到按照属性 A 划分 D 所获得的信息增益为原熵与新熵之差

$$Gain(A)=Info(D)-Info_A(D)$$

信息增益率使用“分裂信息”值将信息增益规范化。分类信息类似于 $Info(D)$，定义为

$$SplitInfo_A(D)=-\sum_{j=1}^{v}\frac{|D_j|}{|D|}\times\log_2(\frac{|D_j|}{|D|})$$

由此定义信息增益率为

$$GainRatio(A)=\frac{Gain(A)}{SplitInfo(A)}$$

通过上述计算，最终选择具有最大信息增益率的属性作为分裂属性。

```python
# 计算熵（entropy）
def calculate_entropy(dataset, classifier):
    ones = count_positives(dataset.rows, dataset.attributes, classifier)
    total_rows = len(dataset.rows)
    entropy = 0
    p = ones / total_rows # 分为该类的概率
    if (p != 0):
        entropy += p * math.log(p, 2)
    p = (total_rows - ones)/total_rows # 分为另一类的概率
    if (p != 0):
        entropy += p * math.log(p, 2)
    entropy = -entropy
    return entropy
```

(a) 熵

```python
# 计算信息增益
def calculate_information_gain(attr_index, dataset, val, entropy):
    classifier = dataset.attributes[attr_index]
    attr_entropy = 0
    total_rows = len(dataset.rows)
    gain_upper_dataset = csvdata(classifier)
    gain_lower_dataset = csvdata(classifier)
    gain_upper_dataset.attributes = dataset.attributes
    gain_lower_dataset.attributes = dataset.attributes
    gain_upper_dataset.attribute_types = dataset.attribute_types
    gain_lower_dataset.attribute_types = dataset.attribute_types

    for example in dataset.rows:
        if (example[attr_index] >= val):
            gain_upper_dataset.rows.append(example)
        elif (example[attr_index] < val):
            gain_lower_dataset.rows.append(example)

    if (len(gain_upper_dataset.rows) == 0 or len(gain_lower_dataset.rows) == 0):
        return -1

    attr_entropy += calculate_entropy(gain_upper_dataset, classifier) * len
(gain_upper_dataset.rows) / total_rows
    attr_entropy += calculate_entropy(gain_lower_dataset, classifier) * len
(gain_lower_dataset.rows) / total_rows

    return entropy - attr_entropy
```

(b) 信息增益

Figure 1: 信息增益率

1.2 连续型属性的离散化处理

当属性类型为离散型，无须对数据进行离散化处理；当属性类型为连续型，则需要对数据进行离散化处理。
C4.5 算法针对连续属性的离散化处理的核心思想是：

1. 将属性 A 的 N 个属性值按照升序排列，得到属性 A 的属性值取值序列 $x_1^A, x_2^A, \cdots, x_N^A$。
2. 在序列 $x_1^A, x_2^A, \cdots, x_N^A$ 中共有 $N-1$ 种二分方法，产生 $N-1$ 种分隔阈值，对于第 i 种二分方法，其阈值 $\theta_i = \frac{x_i^A + x_{i+1}^A}{2}$，它将该节点上的数据集划分为 2 个子数据集 $x_1^A, x_2^A, \cdots, x_i^A$ 和 $x_{i+1}^A, x_{i+2}^A, \cdots, x_N^A$。
3. 计算每种划分方法对应的信息增益，选取信息增益最大的划分方法的阈值作为属性 A 二分的阈值。

```
    # 需要继续细分
    splitting_attribute = None
    maximum_info_gain = 0 # 最大信息增益
    split_val = None
    minimum_info_gain = 0.01
    entropy = calculate_entropy(dataset, classifier)

    # 对数据的每一列
    for attr_index in range(len(dataset.rows[0])):
        if (dataset.attributes[attr_index] != classifier):
            local_max_gain = 0
            local_split_val = None
            attr_value_list = [example[attr_index] for example in dataset.rows] # 同一特征所有可
能的取值
            attr_value_list = list(set(attr_value_list)) # 去重

            # 数据太多，缩小一下范围
            if(len(attr_value_list) > 100):
                attr_value_list = sorted(attr_value_list)
                total = len(attr_value_list)
                ten_percentile = int(total/10)
                new_list = []
                for x in range(1, 10):
                    new_list.append(attr_value_list[x*ten_percentile])
                attr_value_list = new_list

            # 寻找信息增益最大的特征取值作为根节点
            for val in attr_value_list:
                current_gain = calculate_information_gain(attr_index, dataset,val,entropy)
                if (current_gain > local_max_gain):
                    local_max_gain = current_gain
                    local_split_val = val
            if (local_max_gain > maximum_info_gain):
                maximum_info_gain = local_max_gain
                split_val = local_split_val
                splitting_attribute = attr_index
```

Figure 2: 离散化处理

1.3 剪枝

由于决策树的建立完全是依赖于训练样本，因此该决策树对训练样本能够产生完美的拟合效果。但这样的决策树对于测试样本来说过于庞大而复杂，可能产生较高的分类错误率。这种现象就称为过拟合。
因此需要将复杂的决策树进行简化，即去掉一些节点解决过拟合问题，这个过程称为剪枝。
剪枝方法分为预剪枝和后剪枝两大类。
预剪枝是在构建决策树的过程中，提前终止决策树的生长，从而避免过多的节点产生。预剪枝方法虽然简单但实用性不强，因为很难精确的判断何时终止树的生长。
后剪枝是在决策树构建完成之后，对那些置信度不达标的节点子树用叶子结点代替，该叶子结点的类标号用该节点子树中频率最高的类标记。

```python
# 某组数据是否预测正确
def validate_row(node, row):
    if (node.is_leaf_node == True):
        projected = node.classification
        actual = int(row[-1])
        if (projected == actual):
            return 1
        else:
            return 0
    value = row[node.attribute_split_index]
    if (value >= node.attribute_split_value):
        return validate_row(node.left_child, row)
    else:
        return validate_row(node.right_child, row)
# 数据集预测正确的概率
def validate_tree(node, dataset):
    total = len(dataset.rows)
    correct = 0
    for row in dataset.rows:
        correct += validate_row(node, row)
    return correct/total
# 剪枝
def prune_tree(root, node, validate_set, best_score):
    if (node.is_leaf_node == True): # 叶子节点
        classification = node.classification
        node.parent.is_leaf_node = True
        node.parent.classification = node.classification
        if (node.height < 20):
            new_score = validate_tree(root, validate_set)
        else:
            new_score = 0

        if (new_score >= best_score):
            return new_score
        else:
            node.parent.is_leaf_node = False
            node.parent.classification = None
            return best_score
    else: # 非叶子节点
        new_score = prune_tree(root, node.left_child, validate_set, best_score)
        if (node.is_leaf_node == True):
            return new_score
        new_score = prune_tree(root, node.right_child, validate_set, new_score)
        if (node.is_leaf_node == True):
            return new_score
        return new_score
```

Figure 3: 后剪枝操作

2 K 折交叉验证

将训练集数据划分为 K 部分，利用其中的 $K-1$ 份做为训练，剩余的一份作为测试，最后取平均测试误差做为泛化误差。
K 折交叉验证利用了无重复抽样技术的好处：每次迭代过程中每个样本点只有一次被划入训练集或测试集的机会。

```python
# K 折交叉验证
def KFold():
    training_set = copy.deepcopy(dataset)
    training_set.rows = []
    test_set = copy.deepcopy(dataset)
    test_set.rows = []
    validate_set = copy.deepcopy(dataset)
    validate_set.rows = []

    K=10
    # Stores accuracy of the 10 runs
    accuracy = []
    # start = time.clock()
    for k in range(K):
        print("Doing fold ", k)
        training_set.rows = [x for i, x in enumerate(dataset.rows) if i % K != k]
        test_set.rows = [x for i, x in enumerate(dataset.rows) if i % K == k]
        root = compute_decision_tree(training_set, None, classifier)
        results = []
        for instance in test_set.rows:
            result = get_classification(instance, root, test_set.class_col_index)
            results.append(str(result) == str(instance[-1]))

        # Accuracy
        acc = float(results.count(True))/float(len(results))
        print("accuracy: %.4f" % acc)
        accuracy.append(acc)
        del root
    mean_accuracy = math.fsum(accuracy)/10
    print("Accuracy  %f " % (mean_accuracy))
```

Figure 4: K 折交叉验证函数

3 可视化决策树

由于我个人的决策树实现不能像 sklearn 库中的那么完美，最终成型的决策树过于庞大，但是经过 10 折交叉验证，准确率有 83%

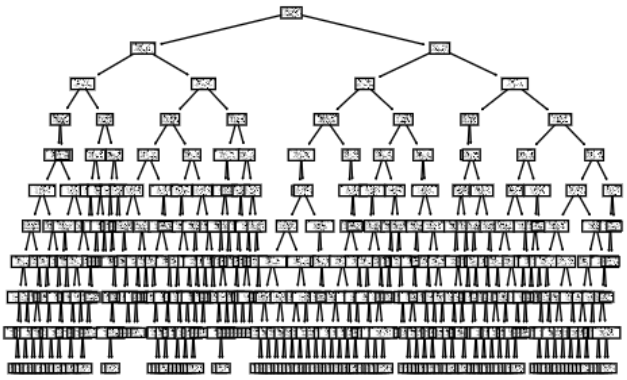

Figure 5: 可视化决策树

结论与分析

C4.5 决策树算法优缺点分析

优点：

1. 通过信息增益率选择分裂属性，克服了 ID3 算法中通过信息增益倾向于选择拥有多个属性值的属性作为分裂属性的不足。
2. 能够处理离散型和连续型的属性类型，即将连续型的属性进行离散化处理。
3. 构造决策树之后进行剪枝操作。
4. 能够处理具有缺失属性值的训练数据

缺点：

1. 算法的计算效率较低，特别是针对含有连续属性值的训练样本时表现的尤为突出。
2. 算法在选择分裂属性时没有考虑到条件属性间的相关性，只计算数据集中每一个条件属性与决策属性之间的期望信息，有可能影响到属性选择的正确性。