中山大学移动信息工程学院本科生实验报告

课程名称:*Artificial Intelligence*

年级	**1501**	专业（方向）	移动互联网
学号	**15352015**	姓名	曹广杰
电话	**13727022190**	**Email**	1553118845@qq.com

一、实验题目

决策树

二、实现内容

实现ID3, C4.5, CART三种决策树

算法原理

决策树:

决策树是根据当前的训练集多个属性概率进行分析，以实现对新的属性集的结果预测的算法。

- **决策树算法的现状**:

  目前来看，决策树的主要作用还是用于实现分类，由于决策树是对已知诸多属性的数据进行分析（这些数据都是有准确答案的），所以决策树算法目前还是在监督学习的范畴内。决策树会根据给出的不同属性，按照树的数据结构进行建模，以便对传入的新的属性集实施预测。

- **决策树的建模**:

  上文说到，决策树会对传入的多个属性以及结果进行分析，具体而言就是:

  1. 根据特定的指标对当前节点，将要进行分类操作会使用的属性进行选择
  2. 根据选定这一属性的种类，确定该节点下分支的实现

  在实现第一点的时候，我们引入了指标——熵（entropy），这种指标将会在每一个节点上作用。

  然而事实上，如果给定一个数据集，每一个label对应的多个属性各自不同，我们完全可以设计出一个树状分类符合训练集中的每一个数据——如果这么实现，决策树的讨论不就终结了吗？

  但是考虑到:

  1. 避免算法过拟合的问题
  2. 排除训练数据集中的噪声干扰

  我们构建的决策树并不追求训练集上的准确率，而是使用另外的衡量指标（本次实验中是熵）作为决策树构建的依据——这个指标的使用使得决策树成为**判断数据分类的趋势**的算法，这样的算法才可以用于新数据集的预测。

ID3:

- 决策策略：信息增益策略

已知:

- 熵代表不确定性，即结果集合的混杂程度。
- 每添加一个属性的限定，混杂程度就会有所改变。

ID3希望总是找出使得当前集合的混杂程度最小的属性，这里混杂程度降低的改变量就称为信息增益。

根据熵的计算公式，如果当前的集合为D，其中的任意一个元素用d表示：

$$H(D) = -\sum_{d \in D} p(d) log_2(p(d))$$

该公式表现了在使用二进制表示该集合的时候，有多少种可能，并将其作为混乱程度的衡量。

而在引入的新的条件（这里拟作是A）之后，混乱程度为：

$$H(D/A) = \sum_{a \in A} p(a) H(D/A = a)$$

讲道理这时候熵应该有所减少。

于是计算熵的变化量——信息增益，不用加绝对值符号：

$$g(D/A) = H(D) - H(D/A)$$

信息增益越大，我们认为该属性对于分类操作具有更好的分辨能力，应该及早使用。故根据ID3算法，我们总是选择可以得到信息增益最大的属性。

C4.5:

- 决策策略：信息增益率

ID3总是选择对集合分类影响最大的属性，但是在该属性分支过多的时候，就会引起过拟合现象。C4.5在此基础上进行了改进，C4.5依然基于熵的衡量标准，不同的是考虑了该属性的熵：

$$\begin{aligned} gRatio(D, A) &= g(D/A)/H(A) \\ &= (H(D) - H(D/A))/\sum_{a \in A} p(a) log_2(p(a)) \end{aligned}$$

很显然，属性的熵与最后的信息增益率呈反比，这就是说，我们希望选择到使用尽量少的分支收获较好分类效果的属性。

CART:

- 决策策略：GINI系数

GINI系数的计算与熵关系不大，在特征A下，数据集D的GINI系数为：

$$\begin{aligned} gini(D, A) &= \sum_{a \in A} p(a) \times gini(D/a) \\ &= \sum_{a \in A} p(a) \times \sum_{i=1}^{n} p_i \times (1 - p_i) \end{aligned}$$

GINI系数作为衡量标准：$\sum_{i=1}^{n} p_i \times (1 - p_i)$表示该数据集中的一个元素已发生概率与未发生概率的乘积。

在其中一方（肯定或者否定）的概率值较大的时候，事件的概率是趋向于这一方的，这时候不确定性较小，乘积也较小；而在两者势均力敌的时候，事件趋向二者的概率相同，不确定性最大，乘积即GINI系数也最大。

GINI系数就是用这种方式表示了当前事件发生概率的不确定性。

伪代码

储存train中的数据的思路如下：

```
存train文件
    建立树状结构( train的属性集、标签集 ){
        //根据信息增益
            => 选择属性集
            => 设选择属性A
        //根据A
            => 分类标签集
            => 分类其他属性集
        递归建立树状结构( 分类后的标签集、属性集 );
    }
```

对valid数据集预测的代码如下：

```
存valid文件
    查询结果( valid的属性集 ){
        //根据当前节点使用的属性
            => 确定应该使用的valid中的属性
            => 设为a
        //根据a的值
            => 选择对应的分支递归查询
            => 传入的参数 ： 【valid属性集 - a】
    }
计算valid预测正确数，返回正确率
```

关键代码截图

为了实现使用决策树对测试样本进行预测分类，需要几个步骤：

1. 设计决策树的数据结构
2. 储存训练集中的数据
3. 根据9个属性进行判断

设计决策树的数据结构

```
class Root{
public:
    Root();
    // 以树的结构储存训练集中的数据
    bool build(vector<vector<int>> Attr_set, vector<int> label);
    int getans(vector<int> attribute);// 根据属性集合返回预测值

    vector<int> label;// 储存的标签集
    vector<int> attr_set;// 标签集对应的属性集
    vector<int> branch_option;// 属性集中去重的元素，作为分支的依据
    vector<Root*> kidergarden;// 不确定数目的分支指针
    int attr_index;// 使用哪个属性划分分支
    int ans;// 如果是叶子节点则有返回值
    bool leafans;// 是否为叶子节点
};
```

笔者使用的树的结构其实就是节点的结构，树即节点，节点即树。在节点中储存数据，将训练集中诸多的信息以树的形式储存起来，以便查询。所以可以看到在声明的节点中，有很多用于储存数据的成员变量。

函数只有两个：build()与getans();

build函数

- 使用的参数：属性集（vector的vector）、标签集（vector）

  使用vector的嵌套以实现二维数组的存储，并可以随机访问

首先要根据信息增益（或者其他的什么）选择属性：

```
    double entropy = 0;
    int NOAttr=0;
    // 遍历属性集，分别测试属性
    for(int atpin=0; atpin<Attr_set.size(); atpin++){
        vector<int> attrset = Attr_set[atpin];// 当前测试的属性信息
      // 使用计算_信息增益_的函数"infogain"
        double crt_epy = infogain(attrset, label);
        if(crt_epy > entropy){  // 总是选择信息增益较大的
            entropy = crt_epy;
            this->attr_set = attrset;
            NOAttr = atpin;// 采用属性 对应的序号
        }
    }
    attr_index = NOAttr;
```

其实这里有很多需要更新数据的地方，就不一一展示了。这里选择好的属性是用一个序号attr_index表示的，在今后的处理中，使用到当前选择的属性时，则根据序号索引获得。

根据选定的属性分类剩余属性以及标签

```
    // 当前节点的属性集
    vector<int> node_attr = Attr_set[NOAttr];
    for(int k=0; k<attr_kind; k++){
        int branch_attr = attr_sample[k];// 获得分支使用的属性

        vector<int> branch_label;
        for(int apin=0; apin<node_attr.size(); apin++){
         // 根据节点属性，分类信息
            if(node_attr[apin] == branch_attr){
                branch_label.push_back(label[apin]);
                /****/
            }
        }
    }
```

内部存在很多分类信息的操作，包括对多个属性储存格式的实现，由于与主题思路关联不大，这里就不予赘述了。

递归建立树状结构

```
        vector<vector<int>> Attr;
        vector<int> branch_label;
        Root *child = new Root();// 新的节点
        this->kidergarden.push_back(child);// 与父节点建立连接
        child->build(Attr, branch_label);
```

已经完成了对属性和标签的分类，将其传入下一个调用函数中即可。

递归查询结果

- 使用参数：属性集合

```
int Root::getans(vector<int> attribute){
    if(leafans == true){
        return ans;
    }
    int crt_attr = Attr_set[NOAttr];// 用于查找的属性
    // 在当前的分支选项中翻找
    for(int brpin=0; brpin<branch_option.size(); brpin++){
        // 找到同样的属性
        if(crt_attr == branch_option[brpin]){
            atr.erase(atr.begin() + attr_index);// 删除这个属性，按照剩下的递归
            return kidergarden[brpin]->getans(atr);// 找到就返回，找不到集继续找！
        }
    }
}
```

实现递归的时候，最重要的环节就是修改传入参数，在这个步骤之前，所有的行为都是在修整参数——删除已经使用过的参数，保留尚未使用的参数。修整参数之后，实现递归调用即可。

创新点与优化

处理新的特征值排列

- 背景：

  由于本次实验仅仅给出了训练集和测试集——没有校验集，笔者在后续调参过程中，手动将训练集根据出现的label次数拆分成两个部分，使得二者的正负label数目大致相同，分别作为校验集和训练集。

- 问题：

  在不剪枝的情况下，由于前期的分支过多，导致后续的分支越来越少，举例说明：

  - 条件：标签对应特征A和特征B，特征B有10种取值。
  - 若从特征A开始实现决策树。
    - 特征A有3种值，第一层分支有3个。
    - 特征B中的10种取值需要分配到3个分支中。
  - 结果：A分出的3个分支中，不能保证**每一个分支都拥有10种取值。**那么，就不能分析特征A和特征B取值的所有排列组合。

  由于校验集中出现了在训练集中为出现过的排列，决策树的查询中不能找到相关的判决，于是抛出错误——一个尚未录入的新的属性，算法不能识别——其实是一个尚未入库的已知特征取值的排列。

- 总结：

  当前的决策树算法需要提高泛化性能。

- 解决方案：

  前提：特征值的大小表示该特征值的程度，即特征值越大，该特征就越明显，反之亦然。

  考虑到在训练集中的特征值排列有$34\times4\times4\times7\times2\times4\times4\times2$种（这里的每一个数字表示该特征值的可能取值），在训练集中的决模型不能完全拟合——完全拟合也没有任何意义，并不能提高泛化性能。

  方案：根据特征值的大小，在当前节点的分支中选择与之最相近的节点，作为决策中该特征值的替代，以便完成决策。

- 实现

```
int near_branch(int num, vector<int> branches){
    // 获得当前分支中的最大值，最小值
    if(num > 最大值){
        return 最小值;
    }else if(num < 最小值){
        return 最小值;
    }else{
        /*
         *遍历序列，找到与num最近的数据 NEAR
         *return NEAR;
         */
    }
}
```

实验去噪

实验中有这样的情况：

- 在某一个节点下，只有一个分支；
- 在这唯一的分支下，只有一个节点；

这种情况在小数据集下是很常见的，但是在有787个条目的时候，这种情况就非常反常了。在787个条目中创建的分支只有一个label，这很显然是为该数据创建的分支，具有很强的数据针对性。

笔者的优化就是要删除这种分支。

检查这种节点：

```
bool Root::build(vector<vector<int>> Attr_set, vector<int> label){
  /***/
    //添加检查节点的操作
  /*
   *如果该分支只有一个分支
   *该分支只有一个节点
   */
    if((Attr_set.size() == 1)&&
        (label.size() == 1)){
        leafans = true;// 确认为叶子节点
        return false;    // 非正常的返回值
    }
}
```

修改这种节点的返回值：

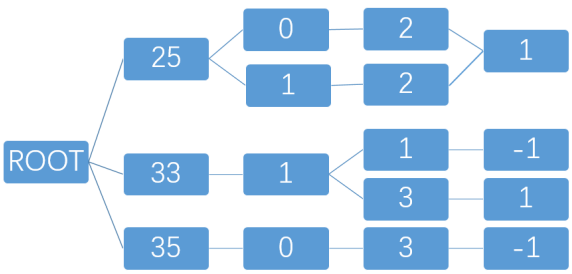
```
        bool team = child->build(Attr, branch_label);
        // 发现了非正常的返回值
        if(team == false){
            int len=this->kidergarden.size();
            if(len > 1)// 将叶子结点中的 label， 修改为相邻节点的label；
                child->ans = this->kidergarden[len-2]->ans;
        }
```

之所以将这种独立节点的label修改为相邻节点的label，就是因为在这个特征值上，临近值与当前分支的特征值最为相近，在排除掉噪点以后，实现合并理应将该分支合并到与之最为相近的分支中。

三、实验结果与分析

1. 实验结果示例展示

   首先设计一个训练集，使之可以形成一个笔者期望的树：



   此后，设计验证集为`25 1 2`和`35 0 3`，可以预见，输出结果分别为`1`和`-1`。

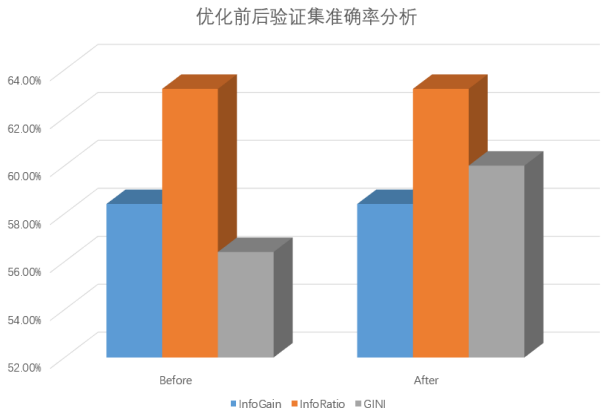
```
The ans=
1
-1
```

小数据集的测试对于三种算法的计算结果是一致的。

2. 评测指标展示及分析

   - 使用数据集：由笔者手动分出训练集与校验集；

优化前后的数据指标分析



可以看到，优化前后对于信息增益以及信息增益率而言都没有什么较大的改变——这是因为在形成决策树的时候，每一个特征的取值都会对应至少2个label——因为在本次实验的训练集中不存在独立节点的特例。

对于GINI指数而言，在优化之前对验证集的准确率是56.4%，优化后是60.0%，可以看到显著的提升。在笔者的测试中发现：在使用GINI指数构建的决策树中，有至少20个这样的独立节点，正是随着这些节点的合并，决策树的泛化性能有了显著的提升。

四、思考

1. **决策树有哪些避免过拟合的方法**

   决策树的过拟合：

   > 为了追求当前数据集的高符合度，使得条件限定得过度严格，以至于不符合其他的数据集。

   决策树的设计在以趋势为主导的时候，即将符合度调节在一个并不太高的程度，使得决策树在进行决策时候不影响其容错率以及泛化性能。

   所谓过拟合就是决策模型更多地针对当前数据集中的一些数据，而这种设计并不符合普适的预测。为此，最有效的方法就是减少对于特定数据的模型构建。

   在减少具有针对性的模型的时候，可以通过预处理、调试时处理或者之后处理的方式进行修整：

   - 预处理：筛选掉训练集中出现的极端值，专注于众数以及平均值
   - 调试的时候：使用前剪枝，去除掉数据量过小的分支，这些分支可能是噪声，总之非主流数据
   - 调试之后：使用后剪枝，针对验证集上的准确率，确定某个分支是否保留

   后剪枝更多的是从实验出发，对理论上的考虑较少，有可能因为对于验证集的拟合，导致在测试集上表现不佳。

2. **C4.5相对于ID3的优势是什么**

ID3考虑当前特征对标签集合的信息增益，C4.5讨论当前特征对标签集合的信息增益率。

后者引入了当前特征的熵的考虑，更加全面——避免了如`编号`这种一一对应的特征对标签分类的影响。对决策模型而言，C4.5使得决策模型更加简化。

3. **如何用决策树判断特征的重要性**

决策树的建立并非追求在训练集上的符合度，而是希望作为数据集中属性相对于label的趋势指示。因此，需要在不同的节点使用不同的特征。

在实现决策树的过程中，我们使用ID3、C4.5以及GINI系数，作为衡量决策树中某层节点使用特征的指标。现在要使用决策树来判断特征的重要性，意为**根据决策树使用某个特征的结果，判断该特征对整个数据集的影响程度**。

一个特征对数据集的影响是否重要，需要考虑几个要素：

- 该特征是否可以最大程度上实现对数据集的分类
- 该特征实现的分类模型是否简洁
- 该特征与将要使用的其他特征之间是否有较强的关联

在不考虑特征之间的关联的时候：

假定所有的特征值之间都是相互独立的，为了在整体趋势上获得对处理的数据集的最好效果的分类，综合考虑前两个因素，可以知道使用C4.5算法在大体上实现了根据趋势的划分。在决策树的表现上就是：**该决策树使用的分支越少，各个分支中的标签信息越纯粹，则该特征越重要**。这种分类的思想是使用最重要的特征进行整体程度上最大的筛选，再由其他的特征值逐步减少分支中的不确定性，最终获得一个较为稳定的决策树结构。

在考虑特征之间的关联的时候：

倘若不同的特征之间有较强的关联性——即某些特征隶属于另一些特征，或者二者在决策的表决上是否有极高的相似度等。这种情况比较复杂，而数学表示的信息又极为有限，只能说，由于不同的特征之间的关联关系，可能会使得在前两个特征上表现较好的特征值不能放在特别靠近root节点的位置。这时候所谓特征的重要性就是无稽之谈——其实这也就揭示了决策树的决策方式其实根本就是来自于不同的判决指标，使用了该特征，则其自然重要，不使用就不重要，脱离了实际的决策模型讨论特征的重要性是没有意义的。