

中山大学数据科学与计算机学院

移动信息工程专业-人工智能

本科生实验报告

(2017-2018 学年秋季学期)

课程名称: Artificial Intelligence

一、实验题目

决策树

二、实验内容

1. 算法原理

(1) 决策树

决策树是一种预测模型，代表的是一种属性和属性值之间的一种映射关系。决策树的每个结点代表一种属性，该结点到子结点的分叉路径代表该属性可能的值；每个叶子结点的值则是从根结点到该叶子结点所经历路径的属性的值对应的结果。当用决策树进行预测时，先从根节点对应的属性开始判断，根据该属性的值确定路径，到达子结点后，再根据子结点对应的属性进行判断，以此类推，直到到达叶子结点，该叶子结点的值就是预测的结果。

决策树的优点：① 分类精度高；② 模型简单，易于理解和实现；③ 对噪声数据有很好的健壮性；④ 能同时处理数据型和常规型属性。

(2) ID3 算法

ID3 算法是决策树的一种。

① 基本思路：

在建立决策树时，用信息增益作为属性的选择标准，选择信息增益最大的属性作为结点，使得在每个非叶子结点进行测试时，能获得关于被测试记录最大的类别信息。然后由该属性的不同取值建立分支。

② ID3 算法中最主要的部分就是条件熵和信息增益的计算。

条件熵用于描述信息不稳定性，条件熵越大，信息越不稳定。其计算公式为：

$$\begin{aligned} & \text{计算特征A对数据集D的条件熵 } H(D|A) \\ H(D|A) &= \sum_{a \in A} p(a) H(D|A=a) \end{aligned}$$

信息增益为经验熵和条件熵的差。

$$\begin{aligned} & \text{计算数据集D的经验熵} & \text{计算信息增益} \\ H(D) &= -\sum_{d \in D} p(d) \log p(d) & g(D, A) = H(D) - H(D|A) \end{aligned}$$

③ ID3 算法的优点:

ID3 算法理论清晰, 算法比比较简单, 学习能力较强, 适于处理大规模的学习问题。

ID3 算法的缺点:

<1> ID3 算法用信息增益作为选择分支属性的标准时, 偏向于取值较多的属性, 而在某些情况下, 这类属性可能并不是最重要的属性。

<2> ID3 算法只能对描述属性为离散型属性的数据集构造决策树, 不能处理具有连续值的属性, 也不能处理具有缺失数据的属性。

<3> ID3 算法虽然理论清晰, 但是计算比较复杂, 会占用较多的内存, 耗费资源。

<4> ID3 是单变量决策树, 在分支结点上只考虑单个属性, 属性间的相关性不够强。

(3) C4.5 算法

C4.5 算法是 ID3 算法的改进。

① 基本思路:

在建立决策树时, 用信息增益率作为属性的选择标准, 选择信息增益率最大的属性作为结点。然后由该属性的不同取值建立分支。

② C4.5 算法中最主要的部分就是熵和信息增益率的计算。

计算数据集D关于特征A的值的熵SplitInfo(D,A)

熵:
$$\text{SplitInfo}(D, A) = - \sum_{j=1}^v \frac{|D_j|}{|D|} \times \log_2 \left(\frac{|D_j|}{|D|} \right)$$

计算信息增益率

信息增益率:
$$\text{gRatio}(D, A) = (H(D) - H(D|A)) / \text{SplitInfo}(D, A)$$

③ C4.5 算法的优点:

<1> C4.5 算法不仅可以处理离散型描述属性的数据集, 还能处理连续型描述属性的数据集或不完整数据。

<2> C4.5 算法是 ID3 算法的改进, 采用了信息增益率作为选择分支属性的标准, 克服了用信息增益来选择属性时变相选择取值多的属性的不足。

<3> C4.5 算法优化了决策树的结构, 提高决策树的生长速度, 避免过度增长和数据过度拟合。

C4.5 算法的缺点:

<1> 在建立决策树的过程中, 需要对数据集进行多次的顺序扫描和排序, 因而导致算法的低效。

<2> C4.5 算法只适用于能够驻留内存的数据集, 当训练集大得无法在内存容纳时程序无法运行。

(4) CART 算法

① 基本思路:

在建立决策树时, 用 GINI 系数作为属性的选择标准, 选择 GINI 系数小的属性作为结点 (GINI 系数越小, 表示不确定性越小)。然后由该属性的不同取值建立分支。

② CART 算法中最主要的部分就是 GINI 系数的计算。



集合 D 的 GINI 系数:

$$Gini(p) = \sum_{k=1}^K p_k(1-p_k) = 1 - \sum_{k=1}^K p_k^2$$

计算特征 A 的条件下, 数据集 D 的 GINI 系数

$$gini(D, A) = \sum_{j=1}^v p(A_j) \times gini(D_j | A = A_j)$$

在属性 A 的特征下, 集合 D 的 GINI 系数:

其中:

$$gini(D_j | A = A_j) = \sum_{i=1}^n p_i(1-p_i) = 1 - \sum_{i=1}^n p_i^2$$

③ CART 算法的优点:

- <1> 能够处理孤立点;
- <2> 能够对空缺值进行处理;

CART 算法的缺点:

CART 本身是一种大样本的统计分析方法, 样本量较小时模型不稳定。

2. 伪代码

```
/*判断是否满足边界条件: 即判断当前结点是否是叶子结点*/
bool meet_with_bound(vector<string> data, vector<vector<string>> attr){
    for(遍历数据集 data){
        统计结果是 label[0]的个数;
    }
    if (label[0]的个数等于数据集的大小 或 label[0]的个数等于 0) {
        data 的样本属于同一个类别, 那么当前结点为叶子结点;
        return true;
    }
    if (特征集 attr 是空集) 当前结点为叶子结点, return true;
    for (从 attr 的第二行开始遍历 attr) {
        将每行 attr 的值和第一行进行对比;
    }
    if (数据集 data 中所有样本在 A 中所有特征上取值相同) 当前结点为叶子结点;
    if (数据集 data 是空集) 当前结点为叶子结点;
}

/*****ID3*****/
/*计算经验熵*/
double get_HD(vector<string> data){
    for (遍历数据集 data) {
        统计 label[0]和 label[1]的数量 num_label1 和 num_label2;
    }
    根据  $H(D) = -\sum_{d \in D} p(d) \log p(d)$  计算经验熵;
}
```



```
/*计算特征集 attr 中每个特征对数据集 data 的条件熵*/
vector<double> getCondition(vector<string> data, vector<vector<string>> attr){
    for（遍历特征集 attr 的每一列）{
        for（遍历特征集 attr 的每一行）{
            if（vector diff_attr 为空）{
                将该特征的取值放到 diff_attr 中，将该值出现的位置记录在 attr_pos;
            }else{
                判断该特征的这个取值是否存在于 diff_attr 中；
                if（存在于 diff_attr 中）{
                    找出该取值在 diff_attr 中的位置；
                    将该取值的位置记录在对应的 attr_pos 中；
                }else {
                    将该取值放到 diff_attr 中，将该值出现的位置记录在 attr_pos;
                }
            }
        }
    }
    for（遍历 diff_attr,即遍历该特征的不同取值）{
        统计在特征为该取值的前提下在 data 中出现 label[0]和 label[1]的个数；
        计算出特征为该值时的  $p(a)H(D|A=a)$ ；
    }
    将该特征的所有取值的  $p(a)H(D|A=a)$ 相加得到条件熵  $H(D|A)$ ：
    
$$H(D|A) = \sum_{a \in A} p(a)H(D|A = a)$$

}
}
```

```
/*计算信息增益*/
vector<double> get_Gain(vector<string> data, vector<vector<string>> attr){
    调用函数 get_HD 获得经验熵；
    for（遍历每个特征的条件熵）{
        根据  $g(D,A) = H(D) - H(D|A)$  计算信息增益；
    }
}

/*****ID3*****/

/*****C4.5*****/

/*计算每个特征下的条件熵和每个特征的熵*/
vector<double> get_cond_split(vector<string> data,vector<vector<string>> attr,vector<double>
&splitInfo){
    for（遍历特征集 attr 的每一列）{
```



<pre>for (遍历特征集 attr 的每一行) { if (vector diff_attr 为空) { 将该特征的取值放到 diff_attr 中, 将该值出现的位置记录在 attr_pos; }else{ 判断该特征的这个取值是否存在于 diff_attr 中; if (存在于 diff_attr 中) { 找出该取值在 diff_attr 中的位置; 将该取值的位置记录在对应的 attr_pos 中; }else { 将该取值放到 diff_attr 中, 将该值出现的位置记录在 attr_pos; } } } for (遍历 diff_attr,即遍历该特征的不同取值) { 统计在特征为该取值的前提下在 data 中出现 label[0]和 label[1]的个数; 计算出特征为该值时的 p(a)H(D A=a); } 将该特征的所有取值的 p(a)H(D A=a)相加得到条件熵 H (D A) : $H(D A) = \sum_{a \in A} p(a)H(D A = a)$ 根据公式 $SplitInfo(D,A) = - \sum_{j=1}^v \frac{ D_j }{ D } \times \log_2 (\frac{ D_j }{ D })$ 计算该特征的熵; }</pre>
<pre>/*计算信息增益率*/ vector<double> get_GainRatio(vector<string> data,vector<vector<string> >attr){ 调用 get_HD 函数获得经验熵 HD; for(遍历每个特征的条件熵) 计算出每个特征的信息增益; 根据公式 $gRatio(D,A) = (H(D) - H(D A))/SplitInfo(D,A)$ 计算信息增益率; } /*****C4.5*****/</pre>
<pre>/*****CART*****/ vector<double> get_gini(vector<string> data,vector<vector<string> > attr){ for (遍历特征集 attr 的每一列) { for (遍历特征集 attr 的每一行) { if (vector diff_attr 为空) { 将该特征的取值放到 diff_attr 中, 将该值出现的位置记录在 attr_pos;</pre>



```
        }else{
            判断该特征的这个取值是否存在于 diff_attr 中；
            if（存在于 diff_attr 中）{
                找出该取值在 diff_attr 中的位置；
                将该取值的位置记录在对应的 attr_pos 中；
            }else {
                将该取值放到 diff_attr 中，将该值出现的位置记录在 attr_pos；
            }
        }
    }
    for（遍历 diff_attr,即遍历该特征的不同取值）{
        统计在特征为该取值的前提下在 data 中出现 label[0]和 label[1]的个数；

        计算出特征为该值时的 $p(A_j) \times gini(D_j|A = A_j)$  ；

    }
    将该特征的所有取值的 $p(A_j) \times gini(D_j|A = A_j)$  相加得到 GINI 系数 gini（D|A）：
    
$$gini(D, A) = \sum_{j=1}^v p(A_j) \times gini(D_j|A = A_j)$$

}
}
/*****CART*****/
/*作用：根据不同的算法选出决策点*/
int choose_attr(vector<string> data, vector<vector<string> > attr,string choose_way){
    if（ID3 算法）{
        调用函数 get_Gain 获得每个特征的信息增益；
        选择信息增益最大的特征作为决策点；
    }
    else if（C4.5 算法）{
        调用函数 get_GainRatio 获得每个特征的信息增益率；
        选择信息增益率最大的特征作为决策点；
    }
    else if（CART）{
        调用函数 get_gini 获得每个特征的 GINI 系数；
        选择 GINI 系数最小的特征作为决策点；
    }
}

/*切割数据集和特征集*/
vector<vector<string> > divide_data(vector<string> data, int choose_attr,
```



```
vector<vector<vector<string>>> &subattr,vector<vector<string>> attr,
vector<string>& diff_attr)
{
    for(遍历该特征 choose_attr 的每一行){
        if (vector diff_attr 为空) {
            将该特征的取值放到 diff_attr 中，将该值出现的位置记录在 attr_pos;
        }else{
            判断该特征的这个取值是否存在于 diff_attr 中;
            if (存在于 diff_attr 中) {
                找出该取值在 diff_attr 中的位置;
                将该取值的位置记录在对应的 attr_pos 中;
            }else {
                将该取值放到 diff_attr 中，将该值出现的位置记录在 attr_pos;
            }
        }
    }
    for (遍历该特征的不同取值) {
        根据 attr_pos 中记录的该取值的行数，将 data 和 attr 切割;
    }
}

/*递归建树*/
void recursive(node *p,string choose_way,vector<string> data , vector<vector<string>> >
attr,vector<bool> & visit ){
    调用 meet_with_bound 函数判断是否满足边界条件
    if (满足边界条件) {
        该结点为叶子结点;
        for(遍历 data){
            统计出现 label[0]和 label[1]的数量;
        }
        选择出现次数多的类别 label 作为该叶子结点的 type;
    }else{
        调用 choose_attr 函数选择出一个特征作为决策点;
        调用 divide_data 函数将数据集 data 和特征集 attr 根据选出的特征的不同取值进行
        切割并且获得该特征的不同取值;
        该结点的孩子结点的数量=切割出来的子数据集的个数;
        该结点的路径的取值=该特征的不同取值;
        for(创建孩子结点){
            调用 recursive，递归建树;
```

<p>将孩子结点放入到该节点的 children 中;</p> <pre> } } }</pre>
<p>/*运用决策树来预测样本的类别*/</p> <pre> vector<string> get_res(node *p1, vector<string> tv){ node *p = p1; for (遍历特征集 tv) { while(p 有孩子结点){ for (遍历路径的 type) { 找到该节点对应的特征的取值对应的路径; p=对应路径的孩子结点; } if(找不到路径){ 将该结点的 type 定义为该样本的类别; 跳出 while 循环; } } if (p 没有孩子结点) 将该叶子结点的 type 定义为该样本的类别; p=p1; } }</pre>

3. 关键代码截图（带注释）

该实验实现了运用不同的算法来建立决策树，然后用决策树去预测样本的类别。

具体方法：检测所有的属性，根据不同的选择标准（ID3 选择信息增益最大的属性，C4.5 选择信息增益率最大的属性，CART 选择 GINI 系数最小的属性）选择某个属性产生决策树结点。由该属性的不同取值建立分支，再对各分支的子集递归调用该方法建立决策树结点的分支，直到所有子集仅包含同一类别的数据或子集为空集或子集对应的特征集的取值全部相同为止。最后得到一颗决策树。

（1）定义结点的结构，并且读取文件，将文件中的数据集记录在 dataset 中，特征集记录在 attribute 中，并且找出数据集中的不同取值放到 label 中。

```

struct node {
    int pos; //第pos个特征
    int n_children; //子结点个数 即特征有几类
    string type; //如果是叶结点，记录label
    vector<string> children_type; //如果有子结点，则记录该特征有几种不同的取值
    vector<node*> children; //孩子结点
};
```




```
node *root; //根结点
vector<string> dataset; //数据集
vector<string> label; //数据集中的不同值
vector<vector<string>> attribute; //存放属性
```

(2) 递归建树:

<1> 首先调用 meet_with_bound 判断是否满足边界条件:

① 如果数据集中的样本属于同一类别 C, 则将当前结点标记为 C 类的叶子结点。

```
int data_size=data.size(),num_label1=0;
for(int i=0;i<data_size;i++){
    if(data[i]==label[0]) num_label1++;
}
if((num_label1==data_size) || (num_label1==0)){ //证明只有一个类别
    return true;
}
```

② 如果特征集 attr 为空集, 则当前结点为叶子结点。

```
if(attr.empty()) return true;
```

如果 data 所有样本在特征集中所有特征上的取值相同, 则当前结点为叶子结点。

```
int attr_size=attr[0].size(), row=1,column=0;
for(int i=1;i<data_size;i++){
    //将attr中的每一行和第一行的attr进行比较
    for(int j=0;j<attr_size;j++){
        if(attr[i][j]==attr[0][j]) column++;
    }
    if(column==attr_size) row++;
}
//如果所有样本的所有特征的取值都相同, 则当前结点为叶子结点
if(row==data_size) return true;
```

③ 如果数据集 data 是空集, 则将当前结点标记为叶子结点。

```
if(data.empty()) return true;
```

<2> 如果满足边界条件, 则该结点为叶子结点, 那么选择数据集中出现次数最多的类别为叶子结点的类别。

```
if(meet_with_bound(data,attr))
{
    /*如果满足边界条件, 则选择类别为dataset中出现最多的类*/
    int data_size=data.size(),num_label1=0,num_label2=0;
    for(int i=0;i<data_size;i++){
        if(data[i]==label[0]) num_label1++;
        else if(data[i]==label[1]) num_label2++;
    }
    if(num_label1>num_label2){
        p->type=label[0];
    }else{
        p->type=label[1];
    }
    p->n_children=0;
    p->pos=-1;
    return;
}
```

<3> 如果不满足边界条件, 那么该结点不是叶子结点。计算出该结点对应的数据集中出现次数最多的类别为该结点的 type。然后根据不同的选择标准选出一个特征作为决策结点。

```
/* choose the best attribute */
int attr_chosen = choose_attr(data,attr,choose_way);
```



ID3 算法:

◆ 首先计算经验熵:

遍历数据集，统计数据集中不同类别出现的次数。

```
for(int i=0;i<data_size;i++){
    if(data[i]==label[0]) num_label1++;
    else if(data[i]==label[1]) num_label2++;
}
```

根据公式 $H(D) = -\sum_{d \in D} p(d) \log p(d)$ 计算经验熵。

```
double label1_en=0,label2_en=0;
if(num_label1==0 || num_label1==data_size) label1_en=0;
else label1_en=(-1)*(1.0*num_label1/data_size)*log(1.0*num_label1/data_size);
if(num_label2==0 || num_label2==data_size) label2_en=0;
else label2_en=(-1)*(1.0*num_label2/data_size)*log(1.0*num_label2/data_size);
double HD=label1_en+label2_en;
```

◆ 计算每个特征的条件熵:

遍历每一种特征，找出每个特征有几种不同的取值，以及这些取值出现的位置。

```
for(int j=0;j<attr_column;j++){ //遍历每一列
    for(int i=0;i<attr_row;i++){ //遍历每一列的每一行
        //遍历找出该特征有几种不同的值，并且记录不同值出现的位置
        if(diff_attr.size()==0){
            diff_attr.push_back(attr[i][j]);
            if(attr_pos.size()==0){
                vector<double> v;
                v.push_back(i);
                attr_pos.push_back(v);
            }
        }else{
            vector<string>::iterator it;
            it=find(diff_attr.begin(),diff_attr.end(),attr[i][j]);
            if(it==diff_attr.end()){
                diff_attr.push_back(attr[i][j]);
                int size_d=diff_attr.size();
                if(attr_pos.size()<size_d){
                    vector<double> v;
                    v.push_back(i);
                    attr_pos.push_back(v);
                }else attr_pos[size_d-1].push_back(i);
            }else{
                int position=distance(diff_attr.begin(),it);
                if(attr_pos.size()<=position){
                    vector<double> v;
                    v.push_back(i);
                    attr_pos.push_back(v);
                }else attr_pos[position].push_back(i);
            }
        }
    }
}
```

根据公式 $H(D|A) = \sum_{a \in A} p(a)H(D|A=a)$ 计算每个特征的条件熵。

```
//计算在该特征下的条件熵
double HDA=0;
int num_diffattr=diff_attr.size(),num_attr=attr.size();
for(int k=0;k<num_diffattr;k++){ //特征的不同值
    int num_pos=attr_pos[k].size();
    int num_label1=0,num_label2=0;
    for(int g=0;g<num_pos;g++){ //计算在特征为该值的前提下出现不同label的数量
        if(data[attr_pos[k][g]]==label[0])
            num_label1++;
        else if(data[attr_pos[k][g]]==label[1])
            num_label2++;
    }

    //计算特征为该值时的条件熵
    double part = 1.0*num_pos/num_attr;
    double label1_en=0,label2_en=0;
    if(num_label1==0 || num_label1==num_pos) label1_en=0;
    else label1_en=(-1)*(1.0*num_label1/num_pos)*log(1.0*num_label1/num_pos);
    if(num_label2==0 || num_label2==num_pos) label2_en=0;
    else label2_en=(-1)*(1.0*num_label2/num_pos)*log(1.0*num_label2/num_pos);
    double cond = 1.0*part*(label1_en+label2_en);
    //将该特征为该值时的条件熵加入到该特征的条件熵
    HDA+=cond;
}
//将该特征的条件熵记录下来
condition_entropy.push_back(HDA);
```



- ◆ 计算每个特征的信息增益：

根据 $g(D, A) = H(D) - H(D|A)$ 计算每个特征的信息增益。

```
double HD=get_HD(data);
vector<double> condition_entropy=getCondition(data,attr);
vector<double> gain;
int con_size=condition_entropy.size();
for(int i=0;i<con_size;i++){
    double diff=HD-condition_entropy[i];
    gain.push_back(diff);
}
return gain;
```

- ◆ 选择信息增益最大的特征为决策结点：

```
if(choose_way=="ID3")
{
    vector<double> gain=get_Gain(data, attr);
    vector<double>::iterator biggest = max_element(gain.begin(), gain.end());
    attr_choose = distance(gain.begin(),biggest);
}
```

C4.5 算法：

- ◆ 计算每个特征下的条件熵和每个特征的熵。

遍历每一种特征，找出每个特征有几种不同的取值，以及这些取值出现的位置。

计算每个特征下的条件熵。

前面两步和 ID3 算法相同。

根据公式 $SplitInfo(D, A) = - \sum_{j=1}^p \frac{|D_j|}{|D|} \times \log_2 \left(\frac{|D_j|}{|D|} \right)$ 计算数据集关于特征 A 的熵。

```
//计算该特征的熵
double split=0,data_size=data.size();
for(int k=0;k<num_diffattr;k++){
    int num=attr_pos[k].size();
    split+=(-1.0)*(1.0*num/data_size)*log(1.0*num/data_size);
}
splitInfo.push_back(split);
```

- ◆ 计算信息增益率。

根据公式 $gRatio(D, A) = (H(D) - H(D|A))/SplitInfo(D, A)$ 计算信息增益率。

```
double HD=get_HD(data);
vector<double> splitInfo;
vector<double> condition_entropy=get_cond_split(data,attr,splitInfo);
//计算信息增益
int con_size=condition_entropy.size();
for(int i=0;i<con_size;i++){
    double diff=HD-condition_entropy[i];
    gain.push_back(diff);
}

//计算信息增益率
vector<double> gainRatio;
for(int i=0;i<con_size;i++){
    double ratio=1.0*gain[i]/splitInfo[i];
    gainRatio.push_back(ratio);
}
```

- ◆ 选择信息增益率最大的特征为决策结点。

```
else if(choose_way=="C4.5")
{
    vector<double> gainRatio = get_GainRatio(data,attr);
    vector<double>::iterator biggest = max_element(gainRatio.begin(), gainRatio.end());
    attr_choose = distance(gainRatio.begin(),biggest);
}
```

CART 算法：

- ◆ 计算每个特征的 GINI 系数。

遍历每一种特征，找出每个特征有几种不同的取值，以及这些取值出现的位置。（和 ID3 相同）



计算特征A的条件下，数据集D的GINI系数

$$gini(D, A) = \sum_{j=1}^p p(A_j) \times gini(D_j | A = A_j)$$

根据其中： $gini(D_j | A = A_j) = \sum_{i=1}^n p_i(1 - p_i) = 1 - \sum_{i=1}^n p_i^2$ 计算 GINI 系数。

```
// 计算在该特征下的gini系数
double GN=0;
int num_diffattr=diff_attr.size(), num_attr=attr.size();
for(int k=0; k<num_diffattr; k++){ // 特征的不同值
    int num_pos=attr_pos[k].size();

    int num_label1=0, num_label2=0;
    for(int g=0; g<num_pos; g++){ // 计算在特征为该值的前提下出现不同label的数量
        if(data[attr_pos[k][g]]==label[0])
            num_label1++;
        else if(data[attr_pos[k][g]]==label[1])
            num_label2++;
    }

    // 计算特征为该值时的gini系数
    double part = 1.0*num_pos/num_attr;
    double gini_part=1.0*part*(1-(1.0*num_label1/num_pos)*(1.0*num_label1/num_pos)-(1.0*num_label2/num_pos)*(1.0*num_label2/num_pos));
    GN+=gini_part;
}
// 将该特征的gini系数记录下来
gini.push_back(GN);
```

◆ 选择 GINI 系数最小的特征为决策结点。

```
else if(choose_way=="CART")
{
    vector<double> gini=get_gini(data, attr);
    vector<double>::iterator smallest = min_element(gini.begin(), gini.end());
    attr_choose = distance(gini.begin(), smallest);
}
```

<4> 用一个 visit 来记录特征是否已经选择。因为选择特征时是在切割后的特征中选择，那么选择的特征的位置和原始的位置是不同的，那么就需要根据 visit 来找到它的原始位置。

```
vector<bool> visit;
for(int i=0; i<attr_num; i++){
    visit.push_back(false);
}

int vi_num=visit.size(), num=-1;
for(int h=0; h<vi_num; h++){
    if(visit[h]==false && num<attr_chosen) num++;
    if(num==attr_chosen){
        visit[h]=true;
        p->pos=h;
        break;
    }
}
```

<5> 根据选择出来的特征的不同取值将数据集和特征集切割成几部分。

```
vector<vector<vector<string>>> subattr;
vector<string> diff_attr;
vector<vector<string>> subsets = divide_data(data, attr_chosen, subattr, attr, diff_attr);
```

切割的详细过程：

◆ 遍历该特征，找出这个特征有几种不同的取值，以及这些取值出现的位置。

```
for(int i=0; i<attr_row; i++){ // 遍历该特征的每一行
    // 遍历找出该特征有几种不同的值，并且记录不同值出现的位置
    if(diff_attr.size()==0){
        diff_attr.push_back(attr[i][choose_attr]);
        if(attr_pos.size()==0){
            vector<double> v;
            v.push_back(i);
            attr_pos.push_back(v);
        }
    }
    else{
        vector<string>::iterator it;
        it=find(diff_attr.begin(), diff_attr.end(), attr[i][choose_attr]);
        if(it==diff_attr.end()){
            diff_attr.push_back(attr[i][choose_attr]);
            int size_d=diff_attr.size();
            if(attr_pos.size()==size_d){
                vector<double> v;
                v.push_back(i);
                attr_pos.push_back(v);
            }
            else attr_pos[size_d-1].push_back(i);
        }
        else{
            int position=distance(diff_attr.begin(), it);
            if(attr_pos.size()-1==position){
                vector<double> v;
                v.push_back(i);
                attr_pos.push_back(v);
            }
            else attr_pos[position].push_back(i);
        }
    }
}
```



- ◆ 遍历不同的取值，将出现该取值的行的数据集和特征集抽出来，形成子数据集和子特征集。

```
for(int i=0;i<row_pos;i++){
    for(int j=0;j<attr_pos[i].size();j++){
        part.push_back(data[attr_pos[i][j]]);
        vector<string> p; //存放子attr的每一行
        for(int k=0;k<column_attr;k++){
            if(k!=choose_attr) p.push_back(attr[attr_pos[i][j]][k]);
        }
        part_attr.push_back(p);
        p.clear();
    }
    subsets.push_back(part);
    subattr.push_back(part_attr);
    part_attr.clear();
    part.clear();
}
```

- <6> 设置该结点的孩子个数，以及路径的取值。

```
int num_sub=subsets.size();
p->n_children=num_sub;
p->children_type.assign(diff_attr.begin(),diff_attr.end());
```

- <7> 调用 recursive 递归建树。

```
for(int i=0;i<num_sub;i++){
    node *subnode = new node();
    vector<bool> vis;
    vis.assign(visit.begin(),visit.end());
    recursive(subnode,choose_way,subsets[i], subattr[i],vis);
    p->children.push_back(subnode);
}
```

- (3) 运用建好的决策树来预测样本的类别。

从根结点开始，判断根节点代表的特征的取值，根据不同的取值选择不同的路径，到达不同的子节点，如果找不到对应的路径，则该结点的类别就是该样本的类别。以此类推，直到到达叶子结点，则该叶子结点的类别就是该样本的类别。

```
vector<string> get_res(node *p1,vector<vector<string> > tv)
{
    vector<string> res;
    int row_tv=tv.size(), column_tv=tv[0].size();
    node *p=p1;
    for(int i=0;i<row_tv;i++){
        while(p->n_children!=0){
            bool have_path=false;
            for(int j=0;j<p->n_children;j++){
                if(tv[i][p->pos]==p->children_type[j]){
                    p=p->children[j];
                    have_path=true;
                }
            }
            if(have_path==false){
                res.push_back(p->type);
                break;
            }
        }
        if(p->n_children==0){
            res.push_back(p->type);
        }
        p=p1;
    }
    return res;
}
```

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

用小数据集来检测建立的决策树是否正确。



	A	B	C	D	E
1	1	1	2	1	0
2	1	1	2	2	0
3	2	1	2	1	1
4	3	2	2	1	1
5	3	3	1	1	1
6	3	3	1	2	0
7	2	3	1	2	1
8	1	2	2	1	0
9	1	3	1	1	1
10	3	2	1	1	1
11	1	2	1	2	1
12	2	2	2	2	1
13	2	1	1	1	1
14	3	2	2	2	0

ID3 算法:

输出经验熵、条件熵、信息增益、选择的特征来判断是否选择是否正确并将树打印出来。

```
经验熵=0.651757
条件熵= 0.480723
信息增益= 0.171034
条件熵= 0.631501
信息增益= 0.0202555
条件熵= 0.546512
信息增益= 0.105244
条件熵= 0.618397
信息增益= 0.0333591
选择第1个特征

经验熵=0.673012
条件熵= 0.277259
信息增益= 0.395753
条件熵= 0
信息增益= 0.673012
条件熵= 0.659167
信息增益= 0.0138443
选择第3个特征

经验熵=0.673012
条件熵= 0.659167
信息增益= 0.0138443
条件熵= 0.659167
信息增益= 0.0138443
条件熵= 0
信息增益= 0.673012
选择第4个特征
```

第1个特征

1

第3个特征

2

1

0

1

2

1

第4个特征

1

1

2

0

C4.5 算法:

输出信息增益、特征的熵、信息增益率以及选择的特征，并将树打印出来

```
信息增益= 0.171034
特征的熵= 1.09337
信息增益率= 0.156428
信息增益= 0.0202555
特征的熵= 1.07899
信息增益率= 0.0187726
信息增益= 0.105244
特征的熵= 0.693147
信息增益率= 0.151836
信息增益= 0.0333591
特征的熵= 0.682908
信息增益率= 0.0488486
选择第1个特征

信息增益= 0.395753
特征的熵= 1.05492
信息增益率= 0.37515
信息增益= 0.673012
特征的熵= 0.673012
信息增益率= 1
信息增益= 0.0138443
特征的熵= 0.673012
信息增益率= 0.0205707
选择第3个特征

信息增益= 0.0138443
特征的熵= 0.673012
信息增益率= 0.0205707
信息增益= 0.0138443
特征的熵= 0.673012
信息增益率= 0.0205707
信息增益= 0.673012
特征的熵= 0.673012
信息增益率= 1
选择第4个特征
```

第1个特征

1

第3个特征

2

1

0

1

2

1

第4个特征

1

1

2

0


CART 算法:

输出特征的 GINI 系数，并将树打印出来。

```
特征的GINI系数= 0.342857
特征的GINI系数= 0.440476
特征的GINI系数= 0.367347
特征的GINI系数= 0.423571
选择第1个特征

特征的GINI系数= 0.2
特征的GINI系数= 0
特征的GINI系数= 0.466667
选择第3个特征

特征的GINI系数= 0.466667
特征的GINI系数= 0.466667
特征的GINI系数= 0
选择第4个特征
```



经过上面的验证，证明建立的决策树是正确的。

接下来把数据集进行切割，分成训练集和验证集。（红色部分为验证集）

	A	B	C	D	E	F
1	1	1	2	1	0	
2	1	1	2	2	0	
3	2	1	2	1	1	
4	3	2	2	1	1	
5	3	3	1	1	1	
6	3	3	1	2	0	
7	2	3	1	2	1	
8	1	2	2	1	0	
9	1	3	1	1	1	
10	3	2	1	1	1	
11	1	2	1	2	1	
12	2	2	2	2	1	
13	2	1	1	1	1	
14	3	2	2	2	0	

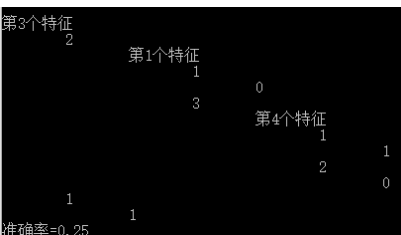
ID3:

```
经验熵=0.673012
条件熵= 0.449868
信息增益= 0.223144
条件熵= 0.52746
信息增益= 0.145552
条件熵= 0.250201
信息增益= 0.42281
条件熵= 0.659167
信息增益= 0.0138443
选择第3个特征

经验熵=0.500402
条件熵= 0.277259
信息增益= 0.223144
条件熵= 0.381909
信息增益= 0.118494
条件熵= 0.381909
信息增益= 0.118494
选择第1个特征

经验熵=0.693147
条件熵= 0.693147
信息增益= 0
条件熵= 0
信息增益= 0.693147
选择第4个特征

准确率=0.25
```



C4.5:

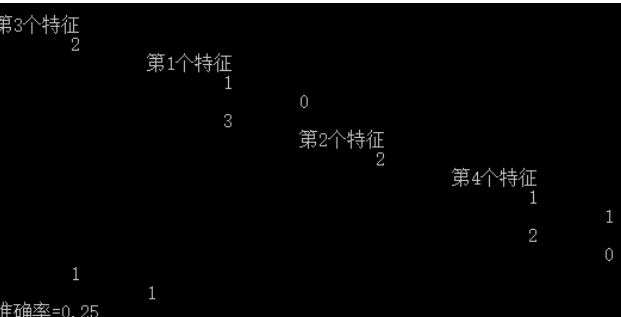
```
信息增益= 0.223144
特征的熵= 1.05492
信息增益率= 0.211526
信息增益= 0.145552
特征的熵= 1.02965
信息增益率= 0.14136
信息增益= 0.42281
特征的熵= 0.693147
信息增益率= 0.609987
信息增益= 0.0138443
特征的熵= 0.673012
信息增益率= 0.0205707
选择第3个特征

信息增益= 0.223144
特征的熵= 0.673012
信息增益率= 0.33156
信息增益= 0.118494
特征的熵= 0.673012
信息增益率= 0.176065
信息增益= 0.118494
特征的熵= 0.673012
信息增益率= 0.176065
选择第1个特征

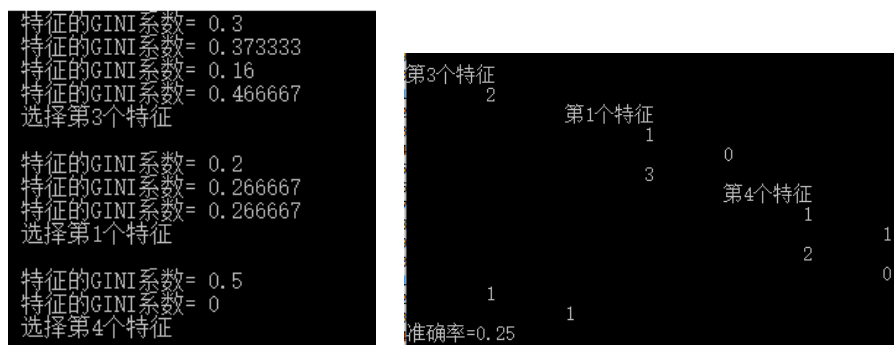
信息增益= 0
特征的熵= 0
信息增益率= nan
信息增益= 0.693147
特征的熵= 0.693147
信息增益率= 1
选择第2个特征

信息增益= 0.693147
特征的熵= 0.693147
信息增益率= 1
选择第4个特征

准确率=0.25
```



CART:



2. 评测指标展示即分析（如果实验题目有特殊要求，否则使用准确率）

ID3: 准确率=0.641221

C4.5: 准确率=0.603053

CART: 准确率=0.637405

四、 思考题

1. 决策树有哪些避免过拟合的方法？

答：决策树避免过拟合的方法：剪枝。

① 预剪枝：

预剪枝是在决策树生成过程中进行。预剪枝通过提前停止树的构建而对树进行剪枝。一旦停止，节点就称为叶子结点，类别为子数据集中出现次数最多的类别。

停止决策树生长的方法有：

- 在划分结点时，根据决策树在验证集上的准确率是否提高来决定是否划分。如果准确率没有提高，则无需划分，直接将当前结点设置为叶子结点。
- 定义一个高度，当决策树达到该高度时就可以停止决策树的生长。
- 定义一个阈值，当达到某个结点的实例个数小于该阈值时就可以停止决策树的生长。即当结点对应的子数据集的数量小于该阈值时就停止决策树的生长，把当前结点作为叶子结点，然后选择数据集中出现次数最多的类别作为该叶子结点的类别。
- 达到某个结点的实例具有相同的特征向量，即使这些实例不属于同一类别，也可以停止决策树的生长。

② 后剪枝

后剪枝是先生成完整的决策树，在自底向上地对非叶子结点进行考察。

后剪枝的方法：

◆ REP, 错误率降低剪枝

该方法中，可用数据分成训练集和验证集，训练集用来形成决策树，验证集来评估剪枝对这个决策树的影响。



在该方法中将树上的每个非叶子结点作为修剪的候选对象。对于某个非叶子结点，如果把它变成叶子结点（叶子结点的类别为子数据集中出现次数最多的类别），决策树在验证集上的准确率不降低，则将它变成叶子结点。

◆ PEP，悲观错误剪枝

该方法是根据剪枝前后的错误率来判定子树的修剪。

对于一个叶子结点，它覆盖了 N 个样本，其中有 E 个错误，那么该叶子结点的错误率为 $\frac{E+0.5}{N}$ 。如果一颗子树有 L 个叶子结点，那么该子树的误判率为：

$$(\sum E_l + 0.5 * L) / \sum N_l$$

假设将该子树进行剪枝变成叶子结点，其误判个数为 J 也要加上一个惩罚因子，变成 $J+0.5$ 。最终是否应该替换的标准为：

$$E(\text{subtree_err_count}) - \text{var}(\text{subtree_err_count}) > E(\text{leaf_err_count})$$

（被替换子树的错误数-标准差>新叶子错误数）

假设该决策树错误分类的概率为 e （ e 为分布的固有属性，可以通过 $(\sum E_l + 0.5 * L) / \sum N_l$ 统计出来），那么树的误判次数就是伯努利分布，那么该树的误判次数均值和标准差为：

$$E(\text{subtree_err_count}) = N * e$$

$$\text{var}(\text{subtree_err_count}) = \sqrt{N * e * (1 - e)}$$

把子树替换成叶子结点后，该叶子的误判次数也是一个伯努利分布，其概率误判率 e 为 $(E+0.5) / N$ ，因此叶子结点的误判次数均值为：

$$E(\text{leaf_err_count}) = N * e$$

◆ CCP，代价复杂度剪枝

该方法为子树定义了代价和复杂度，以及一个可由用户设置的衡量代价与复杂度之间关系的参数 α 。代价是指在剪枝过程中因子树被叶子节点替代而增加的错分样本，复杂度表示剪枝后子树减少的叶节点数， α 表示剪枝后树的复杂度降低程度和代价之间的关系。

$$\alpha = \frac{R(t) - R(T_t)}{|N_t| - 1}$$

$|N_t|$ ：子树中的叶子结点数。

$R(t)$ ：结点 t 的错误代价， $R(t) = r(t) * p(t)$ 。 $r(t)$ 为结点 t 的错分样本率， $p(t)$ 为落入结点 t 的样本占有所有样本的比例。

$R(T_t)$ ：子树 T_t 的错误代价， $R(T_t) = \sum R(i)$ ， i 为子树 T_t 的叶子结点。

运用该方法时，对于决策树 T 的每个非叶子结点计算 α 值，循环减掉具有最小 α 值的子树，直到剩下根节点。那么可以获得一系列的树 $\{T_0, T_1, T_2, \dots, T_m\}$ ， T_0 为

完整的决策树 T ， T_m 为根节点， T_{i+1} 为 T_i 进行剪枝的结果。在子树序列中，根据误差估计选择最佳决策树。

◆ EBP，基于错误的剪枝

对于每个结点，计算剪枝前和剪枝后的误判个数，若剪枝有利于减少误判，则减掉该结点所在的分支。

具体步骤：

① 计算叶子结点的误判率估计的置信区间上限 U 。（置信区间：指由样本统计量所构造的总体参数的估计区间）。

② 计算叶子结点的预测错误样本数。（叶子结点的预测错误样本数=到达该叶子结点的样本数*该叶子结点的误判率 U ）

③ 判断是否剪枝以及如何剪枝。

计算三种预测错误样本数：计算子树 t 的所有叶子结点预测错误样本数之和 $E1$ ；计算子树 t 被剪枝以叶子结点代替时的预测错误样本数 $E2$ ；计算子树 t 的最大分支的预测错误样本数 $E3$ 。

比较 $E1$ 、 $E2$ 、 $E3$ 。 $E1$ 最小时，不剪枝。 $E2$ 最小时进行剪枝，以一个叶子结点代替 t 。 $E3$ 最小时，用最大分支代替 t 。

2. C4.5 相对于 ID3 的优点是什么？

答：ID3 算法用信息增益作为选择分支属性的标准时，偏向于取值较多的属性，而在某些情况下，这类属性可能并不是最重要的属性。C4.5 算法是 ID3 算法的改进，采用了信息增益率作为选择分支属性的标准，克服了用信息增益来选择属性时变相选择取值多的属性的不足。ID3 算法只能对描述属性为离散型属性的数据集构造决策树，不能处理具有连续值的属性，也不能处理具有缺失数据的属性。C4.5 算法不仅可以处理离散型描述属性的数据集，还能处理连续型描述属性的数据集或不完整数据。

3. 如何用决策树来判断特征的重要性？

答：因为我们在建立决策树时，就是通过不同的选择标准来选出重要的特征作为决策结点，所以决策树根结点的特征最重要，越靠近根结点的结点的特征越重要。