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**

教学班级	**1515**	专业（方向）	**软件工程**
学号	**15352334**	姓名	**吴佳卫**

一、 实验题目

决策树

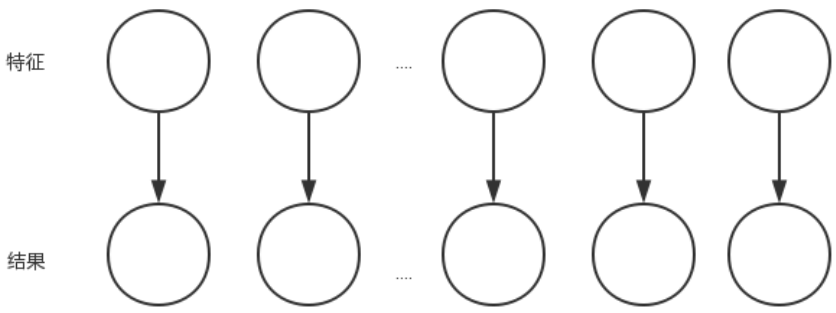
二、 实验内容

1. 算法原理

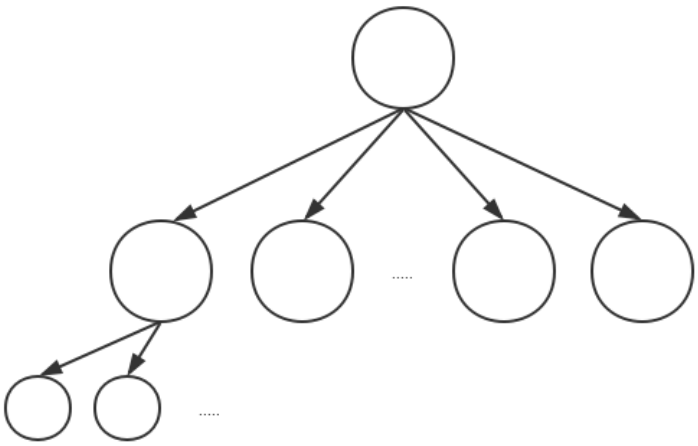
首先说一下个人对决策树的理解。首先，和之前的几个模型一样，决策树是一种有监督机器学习，这也就意味着我们为机器提供了训练样本以及最优模型的评估方法。在得到这个最优的模型以后，我们就可以根据这个模型分析未知的数据。

第二点，决策树算法是一种分类算法，这也就意味着我们会找到某一个训练样本的某一个特征或者是一组特征，然后记录它的结果。在遇到相同或相似的新的样本的时候，我们就可以认为两者是相似的，从而对新的样本作出预测。

那么按照这个理论，决策树应该是以下这个一一对应的模型：



这个例子非常的极端，甚至可以说在训练和验证集上的准确度会非常的高，但是这样一个模型最大的问题就是叶子节点的数目过于的庞大，导致模型的泛化能力非常的低，在测试集的准确度可能非常的低。因此我们引入了“树”的概念以提高模型的泛化能力，如果两个样本有某一些特征是相同的，那我们就可以将两个样本放在一个树的节点下面。



这种做法就随之产生了另一个问题，应该怎样选择位于顶层的节点？最直接的标准就是这个节点对于所有数据的“区分度”。为了衡量“区分度”，我们一共提出了以下三种模型：

1. ID3 模型

$$g(D,A) = H(D) - H(D|A)$$

等式的被减数 H（D），代表 D 的信息熵，也就是对 D 发生的事件的信息量的期望

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

等式的减数，条件熵 H（D|A）表示的是在已知随机变量 A 的条件下，随机变量 D 的不确定性。

$$H(Y \mid X) = \sum_{x \in X} p(x) H(Y \mid X = x)$$

而信息增益就是由信息熵与条件熵来衡量的，对于 D 来说 A 的信息增益大就意味着对于 D 来说，引入 A 使得总的不确定性比较小，因此更加有区分度。

2. C4.5 模型

之所以要引入 C4.5 模型，是因为对于 ID3 模型，如果一个属性的可能的取值比较多，会直接导致信息增益比较大，但是这类的属性可能并不是最有价值的属性。因此我们定义了信息增益率来解决这个问题。

$$\mathrm{GainRatio}_A(D)=\mathrm{Gain}_A(D)/\mathrm{SplitInfo}_A(D)$$

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

分裂信息 SplitInfoA（D）在这里代表按照 A 属性来分割样本 D 的广度，也就是可能会有的取值的数目。然后用信息增益除以分裂信息就得到了信息增益率，可以更好地衡量某一个属性的价值。

3.CART 模型

CART 模型中所使用到的指标是基尼系数，这是一个在经济学中衡量贫富差距的定义，基尼系数越小，贫富差距越小。在这里我们可以这样理解：在某一个特征的条件下，基尼系数越小，代表着不确定性越小，区分度越大。

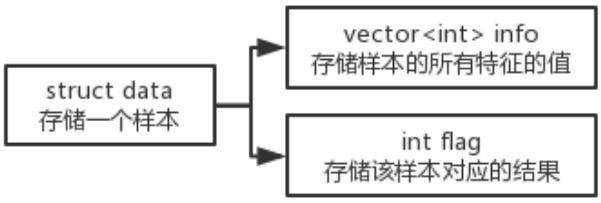
$$\mathrm{gini(D,A)} = \sum_{j=1}^{v} p(A_j)\times gini(D_j|A=A_j)$$

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

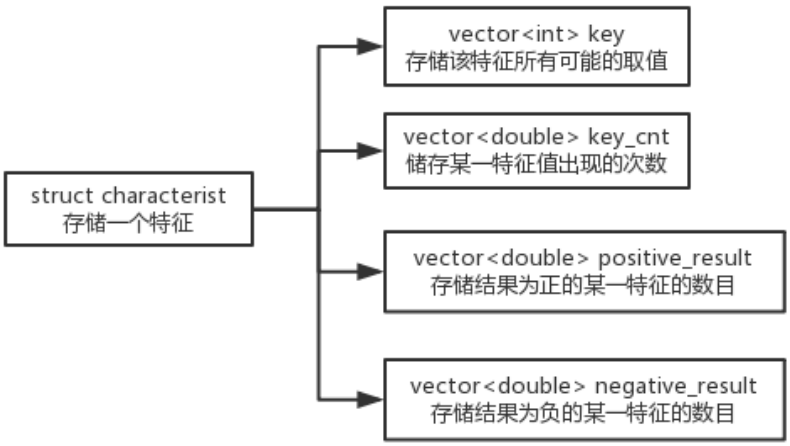
在了解了以上的理论后，我们就得出了建立一颗决策树的方法，也就是每次选出不确定性最小的特征，然后不断地划分训练数据集，知道达到我们所期望的结果。

2. 伪代码

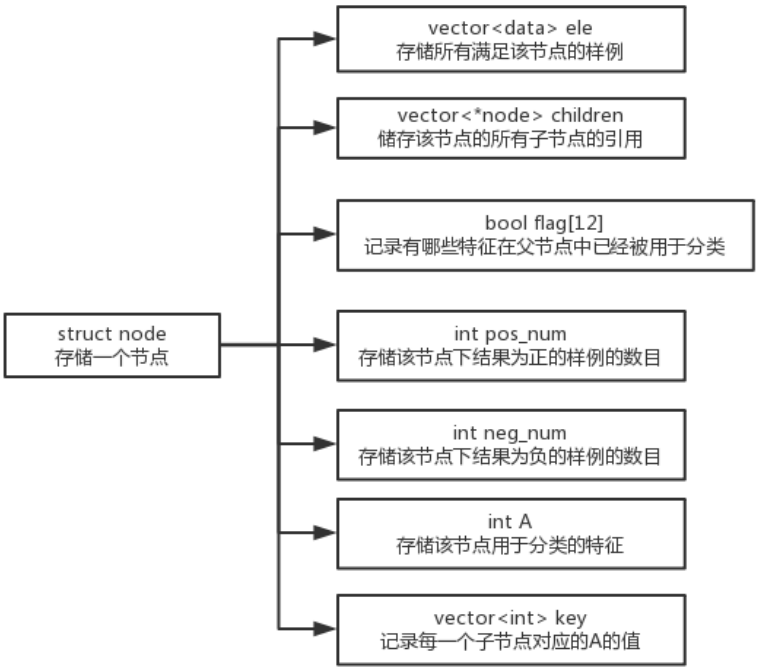
定义了结构体 data 用于存储样本，data 的结构直接按照文件的输入设计。



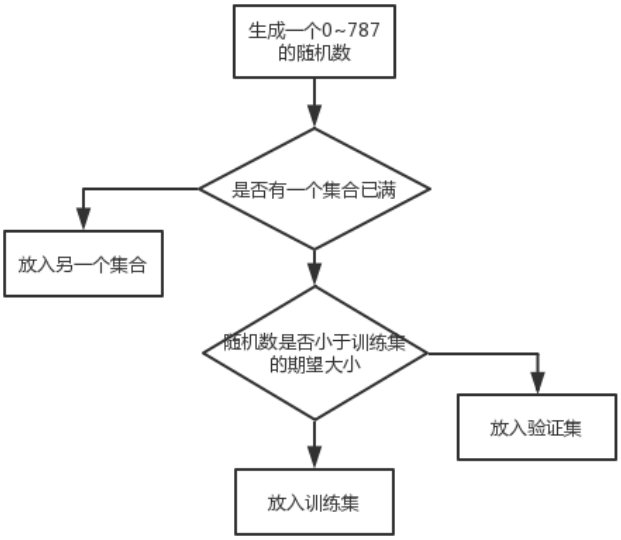
定义了结构体 characterist，这个结构体是针对 vector<data>使用的，目的是辅助我们计算一个数据集合的一些特征。每一个 characterist 存储的是集合中某一个特征的一些属性，从而计算出该特征的信息增益、信息增益率以及基尼指数。



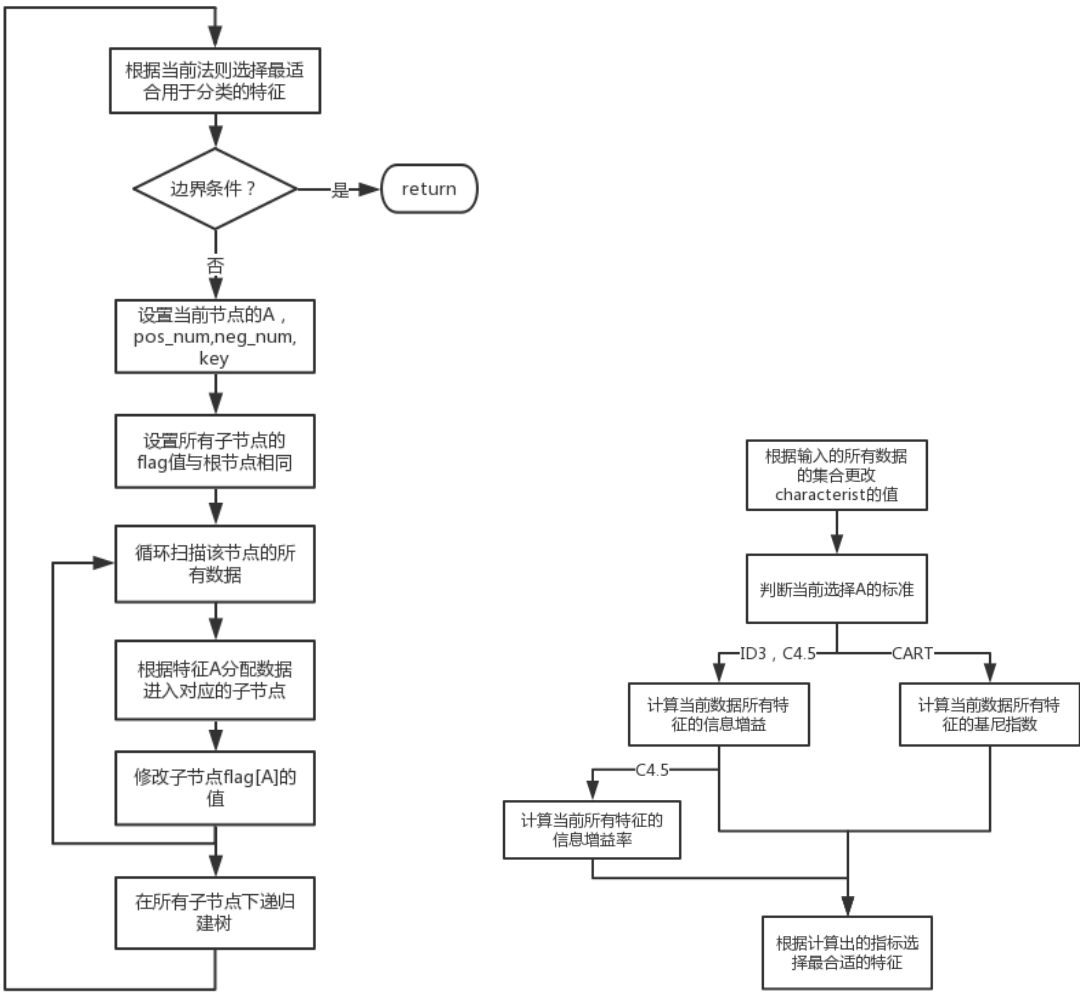
最后定义了结构体 node，也就是决策树的节点。



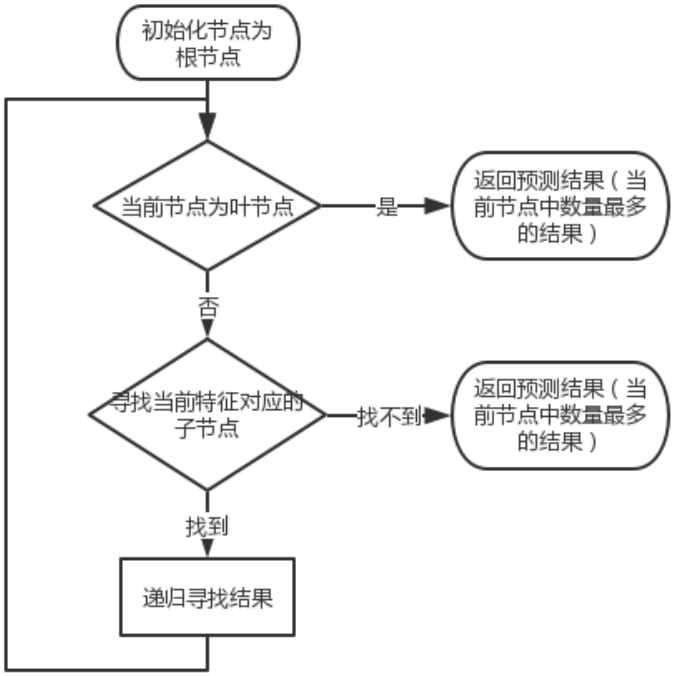
首先需要将集合划分为训练集和验证集 ,在这里我们使用的方法是每次输入一个变量时随机将它放入训练集或验证集。我们只需要定义好训练集的期望大小即可，具体的方法如下。



下面分别是根据当前节点建树的算法以及选择用于分类的特征的算法。



最后是预测测试结果的代码。



3. 关键代码截图

主函数的部分，大概描述了我们用到的函数以及执行过程

```
int type = 1; //0, 1, 2分别对应ID3, C4.5和CART
double percentage_of_train = 0.5; //训练集占输入集合的百分比
train_size = percentage_of_train * 787;
val_size = 787 - train_size;

string train_name = 'DATA\\train.csv';
string test_name = 'DATA\\test.csv';
string result_name = 'DATA\\15352334_wujiawei.txt';

srand (time(NULL)); //初始化时间的种子
data_in(train_name, 0); //输入训练集和验证集
//初始化根节点
for(int i = 0; i < train_set.size(); i ++) root.ele.push_back(train_set[i]);
for(int i = 0; i < 9; i ++) root.flag[i] = true;
//从根节点开始建树
build_tree(&root, type);
cout<<'Build Tree Success!'<<endl;
cout<<'Applying: ';
if(type == 0) cout<<'ID3'<<endl;
else if(type == 1) cout<<'C4.5'<<endl;
else cout<<'CART'<<endl;
val(0); //预测验证集
data_in(test_name, 1); //输入测试集
val(1); //验证测试集
write_result(result_name); //输出预测的测试集结果
return 0;
```

数据处理部分的代码就不做截图了，因为在前几次试验的代码中数据输入以及处理是有重复的部分的。因此这里的截图仅展示随机将输入分配到训练集和验证集的算法。

```
int random_cin() //随机将输入的序列放入验证集或训练集
{
    int a = rand() % 787;
    //如果训练集已满，放入验证集
    if(train_cnt >= train_size) return 1;
    //验证集已满，放入训练集
    if(val_cnt >= val_size) return 0;
    if(a < train_size) return 0;
    else return 1;
}
```

```
if(type == 0 && R == 0) //输入训练集的数据
{
    train_set[train_cnt].info[tmp_dimension] = tmp_num;
}
else if(type == 0 && R == 1) //输入验证集的数据
{
    validation_set[val_cnt].info[tmp_dimension] = tmp_num;
}
```

然后是针对如何选择合适的特征进行建树。在这之前我们首先生成辅助变量 ch[8]代表九种特征的一些属性。

```
double index[10]; //最后衡量每一个特性的指标
memset(index, 0, sizeof(index));
ch.clear();
ch.resize(10);
for(int i = 0; i < 10; i ++)
{
    ch[i].key_cnt.resize(50);
    ch[i].positive_result.resize(50);
    ch[i].negative_result.resize(50);
}
total_positive = 0;
total_negative = 0;
for(int i = 0; i < v.size(); i ++)
{
    for(int j = 0; j < 9; j ++) //更新ch[j]内部的属性
    {
        int tmp_pos;
        vector<int>::iterator it;
        it = find(ch[j].key.begin(), ch[j].key.end(), v[i].info[j]);
        if(it == ch[j].key.end()) //统计属性可能的取值，如果不存在则插入这种取值
        {
            ch[j].key.push_back(v[i].info[j]);
            tmp_pos = ch[j].key.size();
        }
        else tmp_pos = it - ch[j].key.begin();

        ch[j].key_cnt[tmp_pos] ++;
        if(v[i].flag == 1) ch[j].positive_result[tmp_pos] ++;
        else ch[j].negative_result[tmp_pos] ++;
    }
    //计算当前数据集合的一些特性
    if(v[i].flag == 1) total_positive ++;
    else total_negative ++;
}
double total_num = total_negative + total_positive;
```

然后根据需要计算接下来的几个指标，第一个是信息增益。

```
//计算信息熵
double HD = cal(total_positive/total_num) + cal(total_negative/total_num);
double gx[10];
memset(gx, 0, sizeof(gx));

for(int i = 0; i < 9; i ++)
{
    //计算条件熵
    for(int j = 0; j < ch[i].key.size(); j++)
    {
        if(ch[i].key_cnt[j] != 0)
        {
            double tmp1 =  ch[i].positive_result[j] / ch[i].key_cnt[j];
            double tmp2 =  ch[i].negative_result[j] / ch[i].key_cnt[j];
            gx[i] += (ch[i].key_cnt[j] / total_num) * (cal(tmp1) + cal(tmp2));
        }
    }
    index[i] = HD - gx[i]; //计算信息增益
}
```

信息增益率在信息增益的基础上除以分裂信息。

```
for(int i = 0; i < 9; i ++)
{
    //计算分裂信息
    double split_info = 0;
    for(int j = 0; j < ch[i].key.size(); j ++)
    {
        split_info += cal(ch[i].key_cnt[j] / total_num);
    }
    if(split_info != 0) index[i] /= split_info; //计算信息增益率
    else index[i] = 0;
}
```

以及基尼系数。

```
//计算基尼系数
for(int j = 0; j < ch[i].key.size(); j ++)
{
    if(ch[i].key_cnt[j] != 0)
    {
        double tmp1 = ch[i].positive_result[j] / ch[i].key_cnt[j];
        double tmp2 = ch[i].negative_result[j] / ch[i].key_cnt[j];
        index[i] += (ch[i].key_cnt[j] / total_num) * (1 - pow(tmp1, 2) - pow(tmp2, 2));
    }
}
```

根据之前的结果更新当前节点的信息，同时在建树前判断是否达到边界条件。

```
//先更新当前节点内的属性
int A = select_A(N->ele, type, N->flag); //先选择当前分类使用的特征
N->pos_num = total_positive;
N->neg_num = total_negative;
N->A = A;
//如果已经没有可用的分类点，停止建树
if(A == -1) return;
//如果节点下只有一种结果，停止建树
if(N->pos_num == 0 || N->neg_num == 0) return;
//插入并初始化所有的子节点
for(int i = 0; i < ch[A].key.size(); i ++)
{
    N->key.push_back(ch[A].key[i]);
    node* child = new node;
    N->children.push_back(child);
    for(int j = 0; j < 10; j ++) N->children.back()->flag[j] = N->flag[j];
    N->children.back()->flag[A] = false;
}
```

然后开始根据选择的特征来划分数据。并且使用 DFS 递归建树

```
//开始将数据集分类放入子节点
for(int i = 0; i < N->ele.size(); i ++)
{
    vector<int>::iterator it;
    it = find(N->key.begin(), N->key.end(), N->ele[i].info[A]);
    int pos = it - N->key.begin();
    data tmp_data;
    N->children[pos]->ele.push_back(tmp_data);
    for(int j = 0; j < 9; j ++) //插入子节点内的数据集
    {
        N->children[pos]->ele.back().info.push_back(N->ele[i].info[j]);
    }
    //在子节点内记录已经使用过的特征
    N->children[pos]->ele.back().flag = N->ele[i].flag;
}
//DFS递归建树
for(int i = 0; i < N->children.size(); i ++) build_tree(N->children[i], type);
```

最后是如何预测新的数据集，用一个 DFS 来寻找即可。

```
if(N->children.empty()) //如果已经找到叶节点
{
    //如果叶节点存储的正结果多于负，那么预测为正，反之亦然
    if(N->pos_num > N->neg_num) return 1;
    else return -1;
}
else
{
    vector<int>::iterator it = find(N->key.begin(), N->key.end(), D.info[N->A]);
    if(it != N->key.end()) //如果找到了当前特性，DFS继续查询
    {
        return search_tree(N->children[it - N->key.begin()], D);
    }
    else //如果没有找到当前特性的取值，直接在该节点完成预测
    {
        if(N->pos_num > N->neg_num) return 1;
        else return -1;
    }
}
```

三、 实验结果及分析

1. 实验结果展示示例

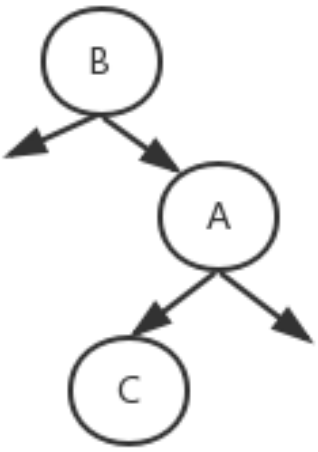
引入小数据集进行测试：

0, 0, 0, 1
0, 0, 1, 1
0, 1, 0, 1
0, 1, 1, −1
1, 0, 0, 1
1, 0, 0, 1
1, 1, 0, −1
1, 0, 1, 1
1, 1, 0, −1

首先使用 ID3 标准进行人工的计算，在根节点的选择上，三个点的信息增益分别为 0.032,0.423,0.004。

然后我们通过观察输出的结果来还原决策树如下：

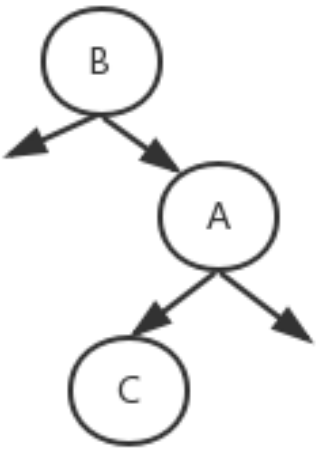
```
0.0321893 0.42281 0.00402174
A= 1
0 0 0
A= -1
back
0.223144 0 0.0505343
A= 0
0 0 0.693147
A= 2
0 0 0
A= -1
back
0 0 0
A= -1
back
0 0 0
A= -1
back
```



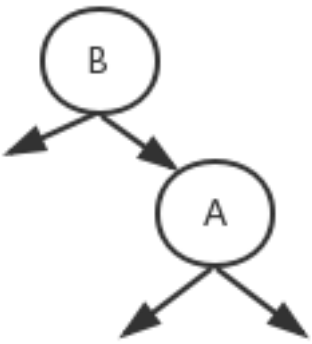
与计算的结果是相同的。

同样的道理，我们可以还原 C4.5 和 CART 算法产生的决策树：

```
0.0478287 0.609987 0.00658369
A= 1
0 0 0
A= -1
back
0.33156 0 0.100987
A= 0
0 0 1
A= 2
0 0 0
A= -1
back
0 0 0
A= -1
back
0 0 0
A= -1
back
```



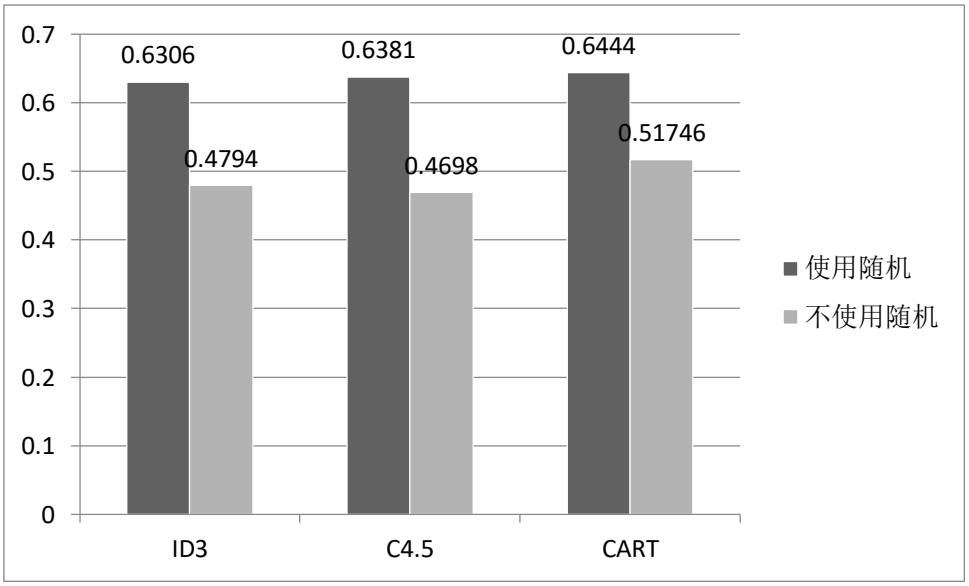

```
0.45 0.16 0.47619
A= 1
0 0 0
A= -1
back
0.2 0.32 0.3
A= 0
0.5 0.5 0
A= -1
back
0 0 0
A= -1
back
```



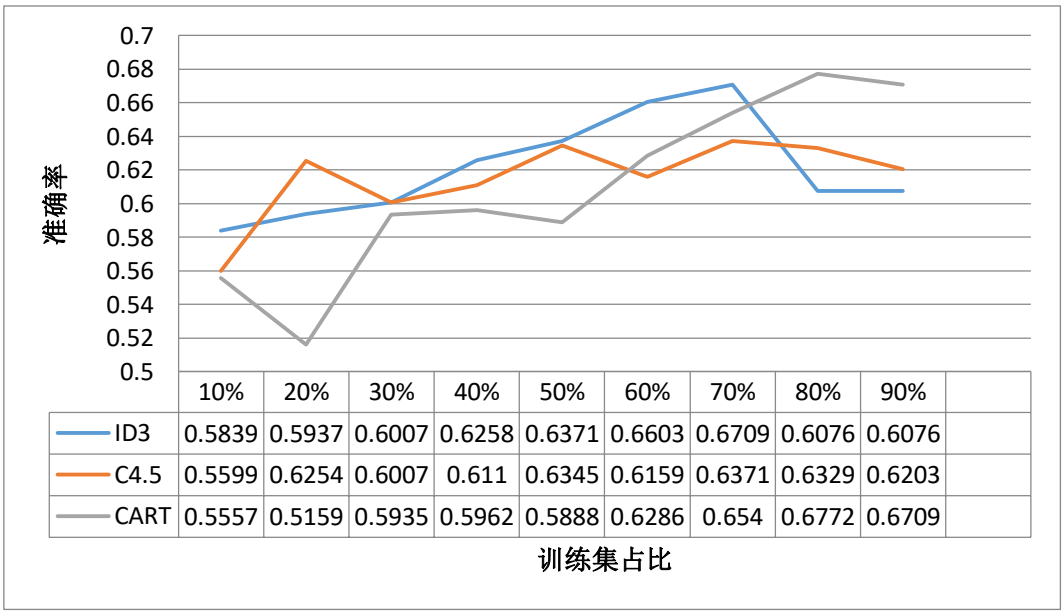
在 CART 算法下，由于计算出 A 节点下 C 的基尼系数为 0，因此没有继续建树。

2. 评测指标展示即分析

这次实验我们一共可以考虑这样两种指标，第一种是读入训练集和验证集的时候是否需要随机输入，以及训练集占的总比重对准确率的影响。首先考虑是否引入随机抽取训练集：



结果是很明显的，因为给定的集合中可能会出现分布不均匀的情况，而决策树作为一种分类算法，要求训练数据集一定是均匀的。再考虑训练集占比的影响：



	10%	20%	30%	40%	50%	60%	70%	80%	90%
ID3	0.5839	0.5937	0.6007	0.6258	0.6371	0.6603	0.6709	0.6076	0.6076
C4.5	0.5599	0.6254	0.6007	0.611	0.6345	0.6159	0.6371	0.6329	0.6203
CART	0.5557	0.5159	0.5935	0.5962	0.5888	0.6286	0.654	0.6772	0.6709

可以观察出随着训练集的数量增加，所有的模型的准确率都呈现上升的趋势，但是受到随机输入的影响都会有一定的波动。但是比较三种决策策略，并没有非常明显可见的不同，只能说从数据上来看 C4.5 的准确率一直比较稳定。

四、 思考题

1. 决策树有哪些避免过拟合的方法？

大概有以下几种避免的方法：

1）降低模型的复杂度：先剪枝，也就是限定决策树的深度，达到深度以后不再构建节点。

2）降低模型的复杂度：后剪枝，判断引入某一些节点以后模型的准确度是否上升。在这里可以引入惩罚项，要求只有准确率提高一定的比例才可以引入一个节点。

3）引入决策森林的概念，也就是在训练决策树的时候通过使用多份随机的训练集生成多个决策树，在预测的时候再根据多个决策树的预测结果来综合得出结论。

2. C4.5 相比于 ID3 的优点是什么？

因为 ID3 是根据信息增益来选择节点的，因此可能会发生这样一种情况：如果有这样的一类特征，它具有很多种可能的取值，但是这些取值出现的频次都非常的低。如果选择这样的特征作为节点会导致决策树的深度大大减小。为了避免选择出这样的特征作为节点。我们将信息增益除以分裂信息，得到了信息增益率。如果某一个特征有非常多的取值，那么分裂信息的值就会变得比较大，信息增益率就会变小。

3. 如何用决策树来判断特征的重要性？

如果某一个特征作为节点比较重要，也就是说明引入这个节点使得不确定性降低。衡量不确定性的标准就是信息增益，信息增益率或者基尼指数。因此一个特征越重要，那么就会越先作为节点被构造决策树。判断特征重要性的方法就是判断这个特征对应的节点是否被优先作为决策树的节点。