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	1508 班	专业（方向）	移动信息工程
学号	15352175	姓名	李其宁

一、 实验题目

决策树

二、 实验内容

1. 算法原理

概述：决策树是离散数学的一种树型表示，可以表示离散函数。相较于 KNN，PLA 等分类算法，其明显的特点是决策的结果可解释性很强。决策树是实现分治策略的数据结构，通过把实例从根节点排列到某个叶子节点来分类实例，可用于分类和回归。

决策树组成：决策树是一种用于对样本进行分类的树形结构。决策树由节点和边组成。节点类型有两种：内部节点个叶子节点。内部节点属于一个特征，节点的分支根据特征的取值来分支；叶子节点即为样本的一个分类。

决策树分类过程：决策树决策是基于树结构来进行决策的，类似与人脑在面临决策问题时一种自然的处理机制：依次判定一个决策问题的属性，到最后得出问题的结论。例如，判断一个西瓜是否为好瓜是，首先判断这个西瓜的颜色，如果颜色为深绿，再来看西瓜的纹理，然后看西瓜的软硬程度。。。最后得出这个西瓜是好的还是坏的。

依据节点纯度选择最优划分属性：在决策树的建树过程中，需要对数据不断划分。一般而言，决策树的分支节点所包含的样本尽可能属于同一个类别，即节点纯度越高越好。在信息论中，期望信息越小，那么信息增益就越大，从而纯度就越高。划分最优属性的方法有 **ID3 信息增益，C4_5 信息增益率，CART 基尼指数；**

ID3(信息增益）：ID3 是通过选择信息增益最大的属性来提高节点的纯度。

- **信息熵**：为离散随机事件出现的概率，信息熵越高，系统越有序；信息熵越低，系统越混乱。一个属性的变化情况越多，它携带的信息量就越大。假定 d 数据的类别，数据 D 的信息熵定义为：

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

- **条件熵**：条件熵是数据集在特征 A 的情况下的信息熵，其定义如下：

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

- **信息增益**：信息增益越大，意味着使用属性 A 来划分所获得的纯度提升越大，因此选

择属性时，选择信息增益最大的属性。信息增益定义如下：

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

C4_5(增益率)：信息增益（**ID3**）对可取值数目较多的属性有所偏好，假如这个属性每一个分支只有一个样本，这些分支结点纯度已经达到最大。然而这种决策树往往导致过拟合，为减少 ID3 这种偏好带来的不利影响，使用增益率来选择最优划分属性。

- **增益率定义：**

$$gRatio(D,A) = (H(D) - H(D|A))/SplitInfo(D,A)$$

其中 SplitInfo(D,A)是在特征 A 的情况下数据集 D 的熵：

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

CART(基尼指数): 基尼指数反映了数据集 D 中随机抽取两个样本，其类别不一致的概率。基尼指数越小，不确定越小，数据集的纯度越高。

- **基尼指数定义：**

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

其中：

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

剪枝处理：决策树考虑了所有数据点而生成的树，因此决策树一个明显的缺点是容易生成过于复杂的树而导致过拟合。一个防止过拟合的方法是剪枝处理，通过剪枝处理降低决策树的复杂度，降低过拟合的概率。

- **预剪枝**：在决策树生成过程中对每一个结点进行估计，若当前结点的划分不能带来决策树泛化性能的提高，则将该结点标记为叶结点。
- **后剪枝**：先生成一个完整的决策数，然后对所有非叶结点进行考察，若将该结点对应的子树替换为叶结点可以提高树的泛化性能，则替换为叶子结点。

验证集划分&对决策树的评估：1 折交叉验证（留一法）

- **k 折交叉验证：**首先将数据集 D 划分为 k 个大小相似的互斥子集（D1，D2，D3，，，Di）。每个子集 Di 尽可能保持数据分布一致性。然后，每次用 k-1 个子集的并集最为训练集，余下的作为验证集；因而可以获得 k 组训练/验证集合，进行了 k 次训练和测试，得到的 k 个验证结果，最终的评估结果是 k 次交叉验证结果的均值。
- **1 折交叉验证：**本次实验对决策树的评估采用 1 折交叉验证来评估。采用这种方法划分验证集和评估决策数的原因如下：

  1、不受随机样本划分方式的影响，m 个样本只能划分 m 个唯一的验证集。

  2、使用的训练集与初始数据集相比至少一个样本，训练数据保留了数据的初始分布。

  3、数据集比较小，如果采样简单划分得到验证集的方法，可能会使训练集丢失某些重

要信息。

- **决策树评估：**采用 1 折交叉验证得到多个训练数据集大小的决策树和训练数据大小的结果，因为每个决策树对一个验证样例判断有正确的和错误的，准确率不是 0 就是 1，因此最终结果是多次价差验证结果的均值。

2. 伪代码

```
Input:训练集  D
      属性集  A
Output：以 root 为根节点的决策树
Process：函数 TreeGenerate(D,A)
1:   生成节点 node；
2:   if D 中的样本属于同一类别 C     then
3:        将 root 标记为 C 类叶节点；  return
4:   end if
5:   if A 为空集，或 D 中所有样本在 A 中所有特征上取值相同
6:        将 root 标记为叶节点，类别标记为 D 中中样本数最多的类：  return
7:   end if
8:   遍历当前节点数据集和特征，根据某种原则，选择一个特征 a
9:   for a  中的每一个值 a_value    do
10:  根据 a_value 的取值，得到 D 中在 a 上取值为 a_value 的样本子集 Di
11:  为 node 生成一个分支
12:  if Di 为空    then
13:       将子节点标记为叶节点，类别为父节点中出现最多的类；  return
14:  else
15:       以 TreeGrnreate（Di，A\{a}）为分支节点。
16:  end if
17:  end for
```

3. 关键代码截图（带注释）

➢ **建树关键代码截图**

- **结点结构**

```
typedef struct node {  //节点
    int label = 0;
    int neg; //当前节点下正例个数
    int pos; //当前节点下负例个数
    int attr;  //节点特征
    vector <int> attr_value;//节点分支依据
    node* children[50] = {NULL}; //有多个孩子
}node;
```

- **读取训练数据，初始化训练集和特征集和根结点。**

```
void init() //initialize the root node and dataset and the attr set
{
    /* build data set */
    fstream fin("train.csv");
    string line;
    int m = 0;
    while(getline(fin,line)){
        istringstream sin(line);
        string field;
        int n = 0;
        while(getline(sin,field,',')){//Data保存数据
            Data[m].push_back(atoi(field.c_str()));//初始化数据集
            if(n!=ATTR_NUM) Attr_value[n].insert(atoi(field.c_str()));//初始化特征集的值
            n++;
        }
        m++;
    }
    length = m;
    for(int i=0;i<ATTR_NUM;i++) attr.push_back(i);//初始化特征集
    set<int> :: iterator it;
}
```

- **递归结束的判断条件**

```
bool meet_with_bound(vector<int> index)
{
    int len = index.size();
    int pos = 0;
    int neg = 0;
    if(len==0)  return true; //当前数据集为空
    for(int i=0;i<len;i++){
        if(Data[index[i]][ATTR_NUM]==1)  pos++;
        else neg++;
    }
    if(neg==0 || pos==0)  return true; //当前数据集标签一致
    if(attr.size()==0) return true; //当前特征集为空
    for(int i=0;i<attr.size();i++){
        int a = Data[index[0]][attr[i]]; //特征集取值相同
        for(int j=1;j<len;j++)  if(Data[index[j]][attr[i]]!=a) return false;
    }
    return true;
}
```

- **选择属性，方法有 ID3，C4_5,CART 三种属性选择方法**

```
int choose_attr(vector<int> index)
{
    if(USE_ID3)
    {
        ID3 id3;
        return id3.getAttr(Data,index,attr);
    }
    else if(USE_C4_5)
    {
        C4_5 c4_5;
        return c4_5.getAttr(Data,index,attr);
    }
    else if(USE_CART)
    {
        CART cart;
        return cart.getAttr(Data,index,attr);
    }
}
```

- 根据属性 a 划分数据集，返回的是样本索引，即索引来表示样本的子集。

```
/* 根据属性 a 分割数据 返回的是样本的一组索引*/
vector<vector<int> > divide_data(int a,vector<int> index)
{
    int col = a;
    vector <vector<int>> v;
    vector <int> indexSet[50];
    vector <vector<int> > v2;
    set<int> :: iterator it;
    int m = 0;
    for(it=Attr_value[col].begin();it!=Attr_value[col].end();it++){
        for(int i=0;i<index.size();i++) if(*it==Data[index[i]][col])  indexSet[m].push_back(index[i]);
        v.push_back(indexSet[m]);
        indexSet[m].clear(); //释放内存
        m++;
    }
    return v;
}
```

- 递归建树

```
void recursive(node *p,vector <int> index)
{
    int attr_chosen = choose_attr(index);//选择最好的属性
    vector<vector<int> > subsets = divide_data(attr_chosen,index);
    p->attr = attr_chosen;
    for(int i=0;i<attr.size();i++) //特征集删除选过的特征
        if(attr[i]==attr_chosen)
            attr.erase(attr.begin()+i,attr.begin()+i+1);
    set<int> :: iterator it;
    //节点分支初始化
    for(it=Attr_value[attr_chosen].begin();it!=Attr_value[attr_chosen].end();it++)
        p->attr_value.push_back(*it);
    it = Attr_value[attr_chosen].begin();
    //节点pos和neg的赋值
    int pos=0,neg=0;
    for(int i=0;i<index.size();i++){
        if(Data[index[i]][ATTR_NUM]==1) pos++;
        else neg++;
    }
    p->pos=pos; p->neg = neg;

  //节点分支和边界的处理
  for(int i=0;i<subsets.size();i++,it++)
  {
      node *subnode = new node;
      /* recursion */
      p->children[*it] = subnode;
      if(meet_with_bound(subsets[i])){//可优化 如果下一个节点是递归结束，下个节点属性则在当前结点处理
          if(subsets[i].size()==0){//下个节点数据集为空
              if(p->pos >= p->neg) subnode->label = 1;
              else    subnode->label =  -1;
              subnode->pos = 0;
              subnode->neg = 0;
              continue;
          }
          else{   //满足递归结束的其他条件，在下个节点处理节点属性
              pos=0,neg = 0;
              for(int j=0;j<subsets[i].size();j++){
                  if(Data[subsets[i][j]][ATTR_NUM]==1) pos++;
                  else neg++;
              }
              if(pos >= neg)  subnode->label = 1;
              else    subnode->label =  -1;
              subnode->pos = pos;
              subnode->neg = neg;
              continue;
          }
      }
      recursive(subnode,subsets[i]);
  }
  attr.push_back(attr_chosen);
  subsets.clear();
  return;
```

- **ID3 属性选择关键代码截图：**

```
/*计算各属性的信息增益*/
double *gain = new double[attr.size()];
for(int k=0;k<attr.size();k++){
    int i = attr[k];
    double attr_pos[50]={0};
    double attr_neg[50]={0};
    for(int j=0;j<len;j++){ //计算各分支的正例与反例的个数
        int a = index[j];
        if(data[a][data[a].size()-1]==1) attr_pos[data[a][i]]++;
        else attr_neg[data[a][i]]++;
    }
    gain[k] = 0;
    //因为计算信息增益的时候，被减数信息熵相同，因此只计算减数，选择减数值越小的的属性
    for(int j=0;j<50;j++) { //计算信息增益
        if(attr_pos[j]!=0 || attr_neg[j]!=0){
            double p_sum = attr_pos[j]+attr_neg[j];
            gain[k]+=p_sum/(len*1.0)*(-xlog2(attr_pos[j]/p_sum)
            -xlog2(attr_neg[j]/p_sum));
        }

    }
}
```

- **C4_5 属性选择关键代码截图：**

```
//计算数据集熵
double H = -xlog2(pos/(len*1.0))-xlog2(neg/(len*1.0));
/*计算各属性的信息增益*/
double *gain = new double[attr.size()];
double *splitinfo = new double[attr.size()];
for(int k=0;k<attr.size();k++){
    int i = attr[k];
    double attr_pos[50]={0};
    double attr_neg[50]={0};
    for(int j=0;j<len;j++){ //计算各分支的正例与反例的个数
        int a = index[j];
        if(data[a][data[a].size()-1]==1) attr_pos[data[a][i]]++;
        else attr_neg[data[a][i]]++;
    }
    gain[k] = 0;
    splitinfo[k] = 0;
    for(int j=0;j<50;j++) { //计算信息增益
        if(attr_pos[j]!=0 || attr_neg[j]!=0){
            double p_sum = attr_pos[j]+attr_neg[j];
            splitinfo[k]+=-xlog2(p_sum/(len*1.0));
            gain[k]+=p_sum/(len*1.0)*(-xlog2(attr_pos[j]/p_sum)
            -xlog2(attr_neg[j]/p_sum));
        }

    }
    //计算增益率
    gain[k]= (H - gain[k])/splitinfo[k];
}
```

- **GINI 属性选择关键代码截图：**

```
/*计算各属性的基尼系数*/
double *gini = new double[attr.size()];
for(int k=0;k<attr.size();k++){
    int i = attr[k];
    double attr_pos[50]={0};
    double attr_neg[50]={0};
    for(int j=0;j<len;j++){ //计算各分支的正例与反例的个数
        int a = index[j];
        if(data[a][data[a].size()-1]==1) attr_pos[data[a][i]]++;
        else attr_neg[data[a][i]]++;
    }
    gini[k] = 0;
    for(int j=0;j<50;j++) { //计算各属性的基尼系数
        if(attr_pos[j]!=0 || attr_neg[j]!=0){
            double p_sum = attr_pos[j]+attr_neg[j];
            gini[k]=p_sum/(len*1.0)*(1-pow(attr_pos[j]/p_sum,2)-pow(attr_neg[j]/p_sum,2));
        }
    }
}
```

4. 创新点&优化（如果有）

PEP 后剪枝（悲观剪枝算法）：本次实验由于用到的是 1 折交叉验证来评估某种方法训练下决策树的性能，单个决策树只有一个验证集。假如采用简单的后剪枝的话，需要使用到一定数量的验证集，对于采用 1 折交叉验证的决策树是无法使用简单的后剪枝的。基于此采用了 PEP 后剪枝，PEP 后剪枝不像 REP（错误率降低修剪）样，需要用部分样本作为验证数据，而是完全使用训练数据来生成决策树，又用这些训练数据完成剪枝。因此本实验采用了 PEP 后剪枝生成决策树，用 1 折交叉验证来评估 PEP 后剪枝生成树的性能。

- **算法原理：**

剪枝后，判定当结点被剪枝后训练集上的误判率为：

$$r(t)=e(t)/n(t)$$

其中：

n(t)为结点 t 中类别 i 的所有样本数；e(t)为 t 中不属于结点 t 所表示类别的样本数，即在结点 t 中误判的个数。

剪枝前，结点 t 的误判率为：

$$r(T_t)=\frac{\sum_{s\in\Omega_{T_t}} e(s)}{\sum_{s\in\Omega_{T_t}} n(s)}$$

其中：

s 为 t 结点下的所有叶子结点

在 PEP 后剪枝把错误分布看成是二项式分布，在逼近正态分布的时候上面的式子有偏差，需

要连续性修正因子来纠正数据，因此有：

$$r\ '(t)=[e(t)+1/2]/n(t)$$

和

$$r'(T_t)=\frac{\sum_{s\in\Omega_{T_t}}[e(s)+1/2]}{\sum_{s\in\Omega_{T_t}}n(s)}$$

因为上述两个式子的分母大小相同，为了方便，使用错误数目而不是错误率，有：

$$e'(t)=[e(t)+1/2]$$

和

$$e'(T_t)=\sum_{s\in\Omega_{T_t}}e(s)+\frac{|\Omega_{T_t}|}{2}$$

接着求 $e'(T_t)$ 的标准差，由于误差近似看成是二项式分布，所以可以得到标准差如下：

$$SE(e'(T_t))=[e'(T_t)\cdot(n(t)-e'(T_t))/n(t)]^{1/2}$$

当结点 t 满足：

$$e'(t)\le e'(T_t)+SE(e'(T_t))$$

则 t 结点则被剪枝。

- **伪代码：**

```
Input：一棵决策树 Tree
Output：标记了哪些结点需要剪枝的决策树 Tree
Process：函数 PEP（node n）;
1：遍历 Tree，计算各个结点剪枝后在训练集上的误判数目 e’(t)：
2：        e’(t) = e(t) + 1/2;
3：if 结点 n 是叶子结点，return  e’(t)
4：else  计算该节点剪枝前在训练集上的误判数目 e’(T)：
5：        遍历结点 n 的所有分支，e’(T) += PEP(结点 n 的所有子孩子)
6:   根据 e’(t)  和 e’(T)判断并标记结点 n 是否应该被剪枝
7：return e’(T)
```

- **关键代码截图：**

➢ **遍历决策树，计算各非叶子结点的 e'(t)和 e'(T)，标记可被剪枝的结点。**

```
double eT(node* n){ //遍历 并标记哪个节点可以剪枝
    if(n->label!=0){ //叶子节点
        double et;//计算叶子节点的et
        if(n->pos > n->neg) et = n->neg + 0.5;
        else if(n->pos < n->neg) et = n->pos + 0.5;
        else et = 0.0;
        return et;
    }
    double et;
    double ET=0.0;
    if(n->pos >= n->neg) et = n->neg + 0.5;
    else et = n->pos + 0.5;  //计算当前节点的et
    for(int i=0;i<50;i++){
        if(n->children[i]!=NULL)
            ET+=eT(n->children[i]); //递归计算当前节点的ET
    }
    double p_sum = n->pos + n->neg;
    double SE = ET * (p_sum-ET)/p_sum;
    //cout<<p_sum<<" "<<ET<<endl;
    SE = sqrt(SE);
    if(et<=ET+SE) n->CUT = 1;
    return ET;
}
```

➢ **剪枝**

```
void cut(node* n){ //剪枝
    if(n->CUT==1){
        if(n->pos > n->neg) n->label = 1;
        else n->label = -1;
    }
}
```

随机森林：

- **算法原理：**使用随机的方式建立一个森林，森林里面有很多决策树组成，随机森林的每一棵决策数之间是没有关联的。在得到森林后，对一个实例的判定首先进过森林中的所有决策树进行判定，然后对所有决策树的判定结果通过投票方式的判断这个样本属于哪一类。森林中的多个决策树由于是随机生成的，因此属于弱决策树，但由于决策是通过全部决策树完成的，因此多个弱的决策树组合成一个强决策树。
- **实现过程：**通过行采样和列采样的方式得到 m 个训练样本和 n 个属性，然后用这些样本和属性训练出一个弱决策树，其中行采样的得到的样本是有放回采样。重复这种步骤多次，得到多个弱决策树，最后将这些弱决策树得到一个强决策树，即随机森林。
- **关键代码截图：**

- **有放回的随机选择样本：**

```
vector<int> select_data(){  //随机选择样本
    vector<int> index;
    for(int i=0;i<sample_num;i++){
        int a = rand()%787;
        index.push_back(a);
    }
    return index;
}
```

- **不放回地随机选择属性**

```
vector<int> select_attr(){ //随机选择属性
    vector<int> attr;
    int attr_vis[9]={0};
    for(int i=0;i<attr_num;i++){
        int a = rand()%9;
        while(attr_vis[a]==1)    a = rand()%9;
        attr_vis[a]=1;
        attr.push_back(a);
    }
    return attr;
}
```

- **构建森林，返回的是一组决策树的根结点**

```
for(int i=0;i<tree_num;i++){ //建树和建森林
    root[i] = new node; //分配空间
    vector<int> index = select_data();
    attr.clear();
    attr = select_attr();
    recursive(root[i],index);
}
```

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）

采用 1 折交叉验证决策树评估性能如下：

- **未剪枝决策树：**

	准确率	精确率	召回率	F 值
ID3	0.63	0.55	0.57	0.56
C4_5	0.63	0.55	0.57	0.56
CART	0.64	0.56	0.58	0.57

- **PEP 后剪枝决策树：**

	准确率	**精确率**	**召回率**	**F 值**
ID3	**0.61**	**0.53**	**0.47**	**0.50**
C4_5	**0.64**	**0.57**	**0.53**	**0.55**

CART	**0.64**	**0.58**	**0.48**	**0.42**

➢ **随机森林（50 棵树，每棵树 300 样本作为训练，选择 4 个属性）：**

	准确率	精确率	召回率	F 值
ID3	**0.64**	**0.57**	**0.51**	**0.54**
C4_5	**0.64**	**0.56**	**0.55**	**0.55**
CART	**0.65**	**0.58**	**0.54**	**0.56**

结论：从以上三个表格可知，

- ID3 模型在四个指标下一般比 C4_5 和 CART 模型的性能低；
- 使用 PEP 后剪枝得到的决策树在 1 折交叉验证上得到的结果与未剪枝的决策树得到结果差不多，有些指标还比未剪枝的低，造成这种结果的原因是：由于 PEP 把错误分布服从二项式分布，在逼近二项式正态分布中，必然带来一些误差。但由于 PEP 后剪枝降低了模型的复杂度，还没有影响决策树的决策性能，因此可以认为 PEP 剪枝后的决策树的性能是比未剪枝的决策树性能强的。
- 随机森林的验证结果普遍比 PEP 后剪枝得到的决策树好，与未剪枝决策树的结果差不多。但由于随机森林避免了过拟合，同时还在验证集上得到了比较好的结果，因此可以判断随机森林的性能比 PEP 剪枝决策树和未剪枝决策树性能好。

2. 小数据集验证

a	b	label
1	1	1
1	1	1
1	0	-1
0	1	-1
-1	1	-1

数据集存在两个属性 a,b,；存在 5 个实例，通过计算我们可以得到数据集的信息熵为：H = 0.9709

➢ **计算 ID3 各属性的信息增益如下：**

G(a) =0.42

G(b) =0.17

单独运行属性选择函数 ID3，运行结果如下：

```
C:\Users\USER\Documents\c++\Untitled1.exe
请选择建树方法 0:ID3 1:C4_5 3:CART
0
当前选择的建树方式：ID3
第0列属性的信息增益为：0.419973
第1列属性的信息增益为：0.170950
选择属性：0

--------------------------------
```

- **计算 C4_5 各属性的信息增益率如下：**

gr(a) = 0.523354

gr(b) = 0.409832

单独运行属性选择函数 C4_5,运行结果如下：

```
C:\Users\USER\Documents\c++\Untitled1.exe
请选择建树方法 0:ID3  1:C4_5  2:CART
1
当前选择的建树方式：C4_5
第0列属性的信息增益为：0.523354
第1列属性的信息增益为：0.409832
选择属性：0

--------------------------------
```

- **计算 CART 各属性的 GINI 系数如下：**

C(a) = 0.27

C(b) = 0.4

单独运行属性选择函数 CART，运行结果如下：

```
C:\Users\USER\Documents\c++\Untitled1.exe
请选择建树方法 0:ID3  1:C4_5  2:CART
2
当前选择的建树方式：CART
第0列属性的信息增益为：0.266666
第1列属性的信息增益为：0.4
选择属性：0

--------------------------------
```

综上所述：由于我们在 ID3、C4_5 和 CART 对属性的选择上，进行了验证，所以可以有理由断定建树过程中建的决策树即是我们想要的树，即建的树是正确的。

四、 思考题

1、决策树有哪些避免过拟合的方法：

- **预剪枝**：在决策树生成过程中对每一个结点进行估计，若当前结点的划分不能带来决策树泛化性能的提高，则将该结点标记为叶结点。

1、**限制生成树的深度**：先预先设定决策树的最大深度，在生成决策树的过程中，若树的深度超过预先设定的阈值，则停止继续向下扩展。

2、**为每个叶子结点设定样本个数的阈值**：生成的决策树的叶子结点只有少量的样

本来用于其决策的话，认为决策树是过拟合的。因此，需预先设定叶子结点的大小，生成决策树后，自底向上判断叶子结点，如果叶子结点小于预定的阈值，则进行剪枝。

3、**为准确率的提升设定阈值：**在生成决策树的过程中，如果在验证集上的准确率超过设定的阈值则停止继续向下生成决策树。

- **后剪枝**：先生成一个完整的决策数，然后对所有非叶结点进行考察，若将该结点对应的子树替换为叶结点可以提高树的泛化性能，则替换为叶子结点。

1、**基于错误率剪枝：**自底向上考察每一个非叶子结点，如果结点剪枝后决策树在验证集的错误有提升，则对该结点剪枝。

2、**基于模型复杂度剪枝：**对一个结点剪枝后，如果准确率的降低小于树复杂度变化的 a 倍时，则对该结点剪枝。

3、悲观剪枝：递归计算决策树内部的结点在训练样本下的误判率，误判率分为剪枝前和剪枝后的，然后通过某个原则比较剪枝前后的误判率来决定该节点是否应该剪枝。

- **随机森林**：随机森林在生成决策树的过程中，由于对数据集和属性集进行采样，并没有全部利用了所有样本的所有数据，避免了过拟合。

2、C4_5 相比与 ID3，有什么优点？

克服 ID3 模型对取值数目较多属性的偏好。考虑一个特征的取值有很多个，并且每个特征的取值下只有一个样本，因此在这种情况下各个叶节点的纯度达到了最大，这也是 ID3 算法所要达到的目的；然而在这种情况下生成的树泛化性能不高，容易导致过拟合。而采用 C4_5 算法可以避免 ID3 的这种缺陷，C4_5 采用启发式方法来选择属性：先从候选属性中找出信息增益高于平均水平的属性，再从中选择增益率最高的。另外，C4_5 算法可以处理连续值，ID3 只能处理离散值。

3、如何用决策树来判断特征的重要性？

- 一个理想的决策树对一个实例进行判断分类时，都是自根结点自顶向下地依次判断。而要最快判断一个实例的类别，首先判断实例的最重要特征，然后按重要性下降方面依次判断各个特征，因此，对应的，在一个理想的决策树中，离根节点越近的特征，其重要程度越大。
- 这次实验属性选择是通过信息增益、信息增益率和基尼指数来划分原样本数据集的，我们选择信息增益和信息增益率大的、基尼指数最小的属性来作为为根结点，然后依次类推选择属性作为子树的根节点。基于上述划分准则可以使一个子树的根结点的子孩子纯度比较高，即各子结点某一类别占的数目明显比其他类别高，也另一个侧面反映了这个特征给决策树提供的信息更多。因此，可以判断，基于 ID3、C4_5 和 GINI 算法生成的决策树，离根结点越近，其重要程度越高。