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	**1511**	专业（方向）	**软件工程（移动信息）**
学号	**15352237**	姓名	**刘顺宇**

一、 实验题目

决策树

二、 实验内容

1. 算法原理

1.1. 简述：决策树是一个树结构（可以是二叉树或多叉树）。其每个非叶节点表示一个特征属性，每个分支代表这个特征属性在某个值域上的输出，而每个叶节点存放一个类别。使用决策树进行决策的过程就是从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为决策结果。决策树这种树形模型是一个一个特征进行处理，层次分明，结构清晰，而之前 PLA 线性模型是所有特征给予权重相加得到一个新的值，所以决策树可以对数据集进行非线性分割，而 PLA 只能做线性分割。

适用问题：非线性分割问题。

1.2. 影响因素：

1.2.1. 决策策略：ID3（信息增益）、C4.5（信息增益率）、CART（GINI 系数）

1.2.2. 剪枝（融于模型复杂度）：

- 预剪枝：在决策树生成过程中对于当前节点计算划分前后的决策树在验证集上的准确率，如果准确率不提高，则无需划分；若当前节点的深度已达限定深度，则将当前节点标记为叶节点，类别为父节点中出现最多的类。
- 后剪枝：生成完整的决策树后用后序遍历对于每个非叶节点，加入将它变成叶子节点，决策树在验证集上的准确率不降低，则将它变成叶子节点。

1.2.3. 递归的边界条件（当前节点数据集为 D，特征集为 A）：

- D 中的样本属于同一类别 C，则将当前节点标记为 C 类节点
- A 为空集，或 D 中所有样本在 A 中所有特征上取值相同，此时无法划分。将当前节点标记为叶节点，类别为 D 中出现最多的类
- D 为空集，则将当前节点标记为叶节点，类别为父节点中出现最多的类。

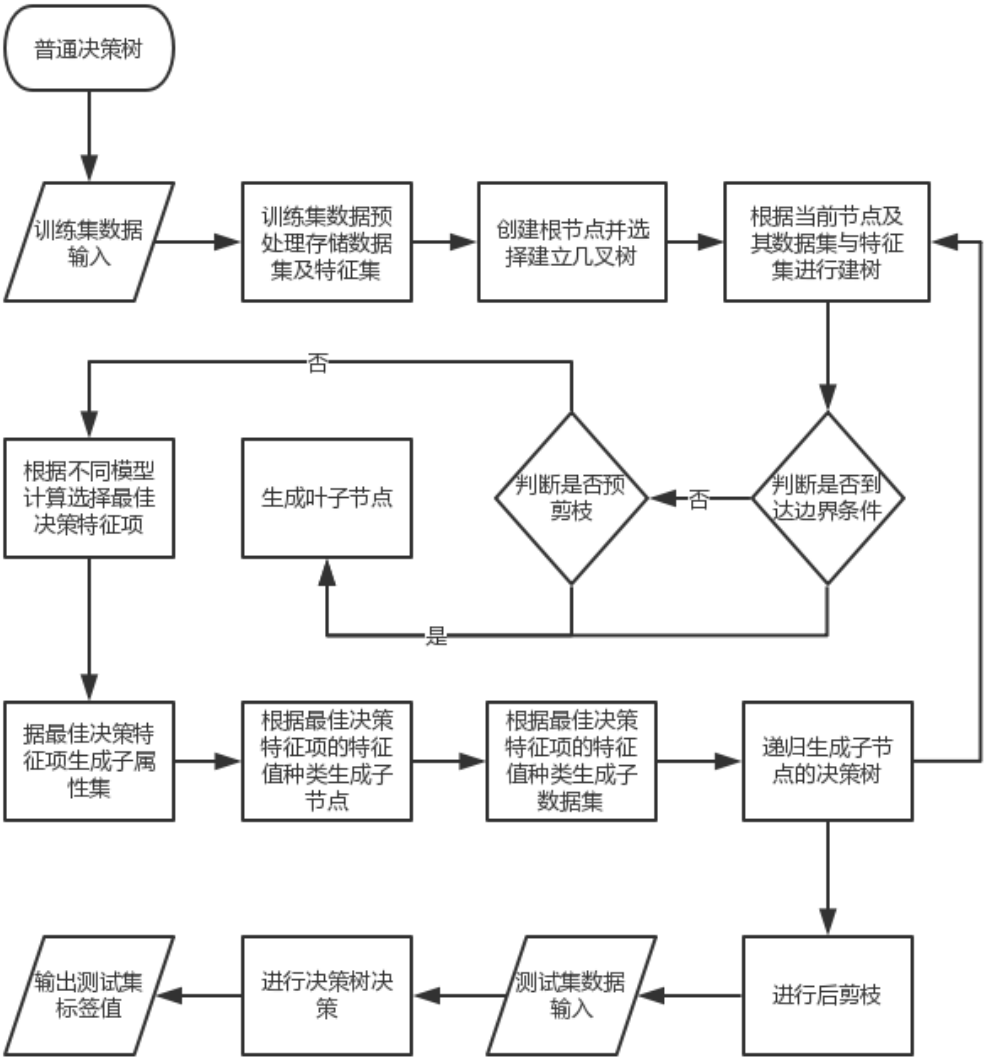
1.2.4. 随机森林：用随机的方式建立一个森林，森林里面有很多的决策树组成，随机森林的每一棵决策树之间是没有关联的，训练时通过随机对行与列进行采样得到数据进行建树。在得到森林之后，当有一个新的输入样本进入的时候，就让森林中的每一棵决策树分别进行一下判断，看看这个样本应该属于哪一类，然后看看哪一类被选择最多，就预测这个样本为那一类。

1.2.5. 叶子节点的基础模型：多数投票法、PLA 线性分割

2. 伪代码流程图

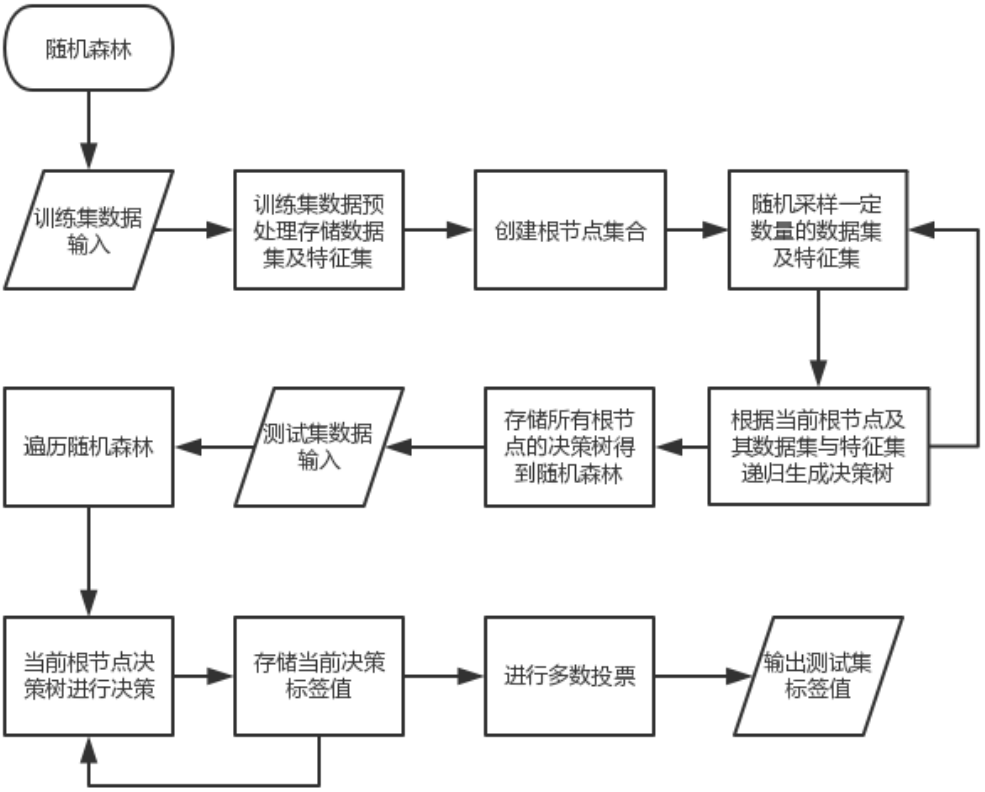
2.1. 普通决策树算法代码流程图（具体代码思想在关键代码解析中讲诉）

首先读取训练集进行处理，然后选用各种不同的决策模型进行建树，再对决策树进行剪枝操作便得到了最后的决策树，接着便能将此决策树运用于测试集数据的决策输出标签值。



2.2. 随机森林

首先读取训练集进行处理，然后通过随机采样一定数量的数据集及特征集进行建树，接着存储当前的决策树，重复上述操作得到随机森林，接着便能将此随机森林运用于测试集数据的决策得到标签值，最后通过多数投票的办法输出标签值。



3. 关键代码截图（这里只解释最关键的建树过程，其他具体代码可看 cpp 文件，有详尽注释）

3.1. **递归建树函数 BuildDecisionTree()**

3.1.1. 三种边界条件判断

第一种：判断是否全部数据集只有一种标签，如果是的话则将当前节点设置为叶子节点，并且直接定义该子节点的标签。

```
//边界判断是否全部数据集只有一种标签
int bound1 = meet_with_bound_OnlyOneLabel(current_data);
if (bound1 != 0) {
    //生成叶子节点
    DecisionTree_node* sub_node = new DecisionTree_node();
    sub_node->property = -1;//定义-1为叶子节点
    sub_node->property_value = bound1;//根据全部数据集只有一种的标签值进行定义
    current_node->child.push_back(sub_node);
    return current_node;
}
```

```
/*
边界判断函数：是否全部数据集只有一种标签
输入：数据集
输出：否或标签值
*/
int meet_with_bound_OnlyOneLabel(vector< vector<int> > current_data) {
    int label = current_data[0][Train_col - 1];
    for (int i = 1; i < current_data.size(); i++) {
        if (label != current_data[i][Train_col - 1])
            return 0;//如果全部数据集不只有一种标签则返回0
    }
    return label;
}
```

第二种 :判断特征集是否为空，如果当前特征集为空则将当前节点设置为叶子节点，并且通过多数投票法来定义当前节点的标签，在这里定义的多数投票函数返回的值不仅是标签值还有具有当前标签值的数据集的数量，这是用于记录叶子节点的错误率，为后剪枝操作做准备。

```
//边界判断特征集是否为空
int bound2 = meet_with_bound_EmptyProperty(current_data, current_property);
if (bound2 != 0) {
    //生成叶子节点
    DecisionTree_node* sub_node = new DecisionTree_node();
    sub_node->property = -1;//定义-1为叶子节点
    if (bound2 > 0)sub_node->property_value = 1;//根据多数投票的结果标签值进行定义
    else sub_node->property_value = -1;
    sub_node->error_num = current_data.size() - abs(bound2 - 1);//记录多数投票结果的错误数
    current_node->child.push_back(sub_node);
    return current_node;
}
```

```
/*
边界判断函数：特征集是否为空
输入：数据集、特征集
输出：否或占比数据集最多的标签值及数量
*/
int meet_with_bound_EmptyProperty(vector< vector<int> > current_data, vector<int> current_property) {
    if (current_property.size() != 0) return 0;
    else return MostLabel(current_data);
}
```

```
/*
数据集标签最多项统计函数
输入：数据集
输出：占比数据集最多的标签值及数量
*/
int MostLabel(vector< vector<int> > current_data) {
    int label_true = 0;//记录数据集中标签为1的数量
    int label_false = 0;//记录数据集中标签为-1的数量
    for (int i = 0; i < current_data.size(); i++) {
        if (current_data[i][Train_col - 1] == 1) label_true++;
        else label_false++;
    }
    if (label_true > label_false) return (label_true + 1);
    else return (-1 - label_false);
}
```

第三种 :判断数据集是否为空，如果当前数据集为空则将当前节点设置为叶子节点，并且通过多数投票法来定义当前节点的标签

```
//边界判断数据集是否为空
int bound3 = meet_with_bound_EmptyData(sub_data);
if (bound3 != 0) {
    //生成子节点的叶子节点
    DecisionTree_node* sub_sub_node = new DecisionTree_node();
    sub_sub_node->property = -1;//定义-1为叶子节点
    if (bound3 > 0)sub_sub_node->property_value = 1;//根据多数投票的结果标签值进行定义
    else sub_sub_node->property_value = -1;
    sub_node->child.push_back(sub_sub_node);
}
```

```
/*
边界判断函数：数据集是否为空
输入：数据集
输出：否或占比数据集最多的标签值及数量
*/
int meet_with_bound_EmptyData(vector< vector<int> > current_data) {
    if (current_data.size() != 0) return 0;
    else return MostLabel(current_data);
}
```

3.1.2. 三种模型选择最佳决策特征项

第一种：ID3 模型，首先计算数据集的经验熵，然后计算特征对数据集的条件熵，最后计算信息增益，并选择信息增益最大的特征作为决策点。

```
/*----------ID3模型----------*/
if (model == 1 || model == 2) {
    int max_Gain_index = current_property[0];
    double p_d = double(cnt_all_true) / double(current_row);
    double H_D = -p_d * (log(p_d) / log(2)) - (1 - p_d) * (log(1 - p_d) / log(2));

    for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
        for (int j = 0; j < 100; j++) {
            if (current_data_cnt[j][*i] != 0 && current_data_cnt_true[j][*i] != current_data_cnt[j][*i]
                && current_data_cnt_true[j][*i] != 0) {
                double p_a = double(current_data_cnt[j][*i]) / double(current_row);
                double p_a_d = double(current_data_cnt_true[j][*i]) / double(current_data_cnt[j][*i]);
                H_D_A[*i] += p_a * (-p_a_d * (log(p_a_d) / log(2)) - (1 - p_a_d) * (log(1 - p_a_d) / log(2)));
            }
        }
        Gain_D_A[*i] = H_D - H_D_A[*i];
        if (Gain_D_A[*i] > Gain_D_A[max_Gain_index]) max_Gain_index = *i;
    }
```

第二种：C4.5 模型，首先计算特征对数据集的信息增益，然后计算数据集关于特征的值的熵，接着用信息增益除以熵得到信息增益率，选择信息增益率最大的特征作为决策点。

```
/*-------------------------------------------------------C4.5模型-------------------------------------------------------*/
if (model == 2) {
    int max_GainRatio_index = current_property[0];
    int tem_col = 0;
    for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
        for (int j = 0; j < 100; j++) {
            if (current_data_cnt[j][*i] != 0) {
                double p_a = double(current_data_cnt[j][*i]) / double(current_row);
                SplitInfo[*i] += -p_a * (log(p_a) / log(2));
            }
        }
        GainRatio[*i] = Gain_D_A[*i] / SplitInfo[*i];
        if (GainRatio[*i] > GainRatio[max_GainRatio_index]) max_GainRatio_index = *i;
    }

    /*for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
        std::cout << "SplitInfo[" << *i << "]: " << SplitInfo[*i] << "  ";
        std::cout << "GainRatio[" << *i << "]: " << GainRatio[*i] << endl;
    }
    std::cout << "root index: " << max_GainRatio_index << endl;
    std::cout << endl;*/
    return  max_GainRatio_index;
}
```

第三者：CART 模型，计算某特征条件下数据集的 GINI 系数，GINI 系数值越小表示不确定性越小，最后选择 GINI 系数最小的特征作为决策点。

```
/*-------------------------------------------------------CART模型-------------------------------------------------------*/
if (model == 3) {
    int max_Gini_index = current_property[0];
    double Gini_j[100][9] = { 0, };
    for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
        for (int j = 0; j < 100; j++) {
            if (current_data_cnt[j][*i] != 0) {
                double p_a_d = double(current_data_cnt_true[j][*i]) / double(current_data_cnt[j][*i]);
                Gini_j[j][*i] = 1 - pow(p_a_d, 2) - pow((1 - p_a_d), 2);
                double p_a = double(current_data_cnt[j][*i]) / double(current_row);
                Gini[*i] += p_a * Gini_j[j][*i];
            }
        }
        if (Gini[*i] < Gini[max_Gini_index]) max_Gini_index = *i;
    }
    /*for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
        std::cout << "Gini[" << *i << "]: " << Gini[*i] << endl;
    }
    std::cout << "root index: " << max_Gini_index << endl;
    std::cout << endl;*/
    return max_Gini_index;
}
```

3.1.3. 递归生成子节点的决策树

首先选择最佳决策特征项，然后根据最佳决策特征项生成去除了最佳特征项的子特征集，接着我们通过已经存储好的映射函数得到最佳决策特征项的取值种类，并通过不同的取值种类生成子数据集，接着便可以通过子特征值和子数据集递归生成子节点的决策树，当然在此之前还得存储住当前节点的特征与特征值。

可以看到代码中我们是将第三种边界判断放在了递归生成决策树的函数之前的，这样便于我们在生成子数据集后便马上进行判断。

```
//选择最佳决策特征项
int best_property = choose_property(current_data, current_property, model);

//根据最佳决策特征项生成子特征集
vector<int> sub_property;
for (vector <int>::iterator i = current_property.begin(); i < current_property.end(); i++) {
    if (*i != best_property) sub_property.push_back(*i);
}

//得到最佳决策特征项的特征值种类
vector<int> current_property_value = Train_property_value[best_property];

//根据最佳决策特征项的特征值种类生成子节点
for (vector <int>::iterator i = current_property_value.begin(); i < current_property_value.end(); i++) {
    vector< vector<int> > sub_data;
    DecisionTree_node* sub_node = new DecisionTree_node();
    //根据最佳决策特征项的特征值种类生成子数据集
    for (int j = 0; j < current_data.size(); j++) {
        if (current_data[j][best_property] == *i) sub_data.push_back(current_data[j]);
    }
    //定义子节点的特征与特征值
    sub_node->property = best_property;
    sub_node->property_value = *i;

    //边界判断数据集是否为空
    int bound3 = meet_with_bound_EmptyData(sub_data);
    if (bound3 != 0) {
        //生成子节点的叶子节点
        DecisionTree_node* sub_sub_node = new DecisionTree_node();
        sub_sub_node->property = -1;//定义-1为叶子节点
        if (bound3 > 0)sub_sub_node->property_value = 1;//根据多数投票的结果标签值进行定义
        else sub_sub_node->property_value = -1;
        sub_node->child.push_back(sub_sub_node);
    }
    //递归生成子节点的决策树
    else {
        sub_node = BuildDecisionTree(sub_node, sub_data, sub_property, model, depth + 1, BeforePruning);
    }
    current_node->child.push_back(sub_node);
}
return current_node;
```

4. 创新点&优化（为了避免重复，创新点算法没有在关键代码截图中提及）

4.1. 预剪枝

在决策树生成过程中进行，通过限定层数来进行预剪枝。若当前节点的深度已达限定深度，则将当前节点标记为叶节点，类别为父节点中出现最多的类。注意在这里定义的多数投票函数返回的值不仅是标签值还有具有当前标签值的数据集的数量，这是用于记录叶子节点的错误率，为后剪枝操作做准备。

```
//预剪枝判断是否达到了最大深度
if (BeforePruning == true && depth == 5) {
    //生成叶子节点
    int bound4 = MostLabel(current_data);
    DecisionTree_node* sub_node = new DecisionTree_node();
    sub_node->property = -1;//定义-1为叶子节点
    if (bound4 > 0)sub_node->property_value = 1;//根据多数投票的结果标签值进行定义
    else sub_node->property_value = -1;
    sub_node->error_num = current_data.size() - abs(bound4 - 1);//记录多数投票结果的错误数
    current_node->child.push_back(sub_node);
    return current_node;
}
```

4.2. 后剪枝

先生成完整的决策树，再通过后序遍历自底向上地对非叶节点进行考察，对于某个非叶节点，假如将它变成叶子节点，判断决策树在训练集上的基于模型复杂度的错误率是否降低，从而判断是否将它变成叶子结点。

在比较错误率是否降低前我们需要得到两个参数分别是决策树的叶子节点数和错误数，这个在前面建树中已经通过一定的结构存储住了需要参数，而且还定义了两个遍历决策树得到当前节点下的决策树的叶子节点数和错误数的函数，让我们可以计算剪枝后的模型复杂度。

```
//计算剪枝后的融入模型复杂度的错误率
int new_error_num = 0;//记录错误数
int new_label = 0;//记录叶子节点数
if (current_node->label_true_num > current_node->label_false_num) {
    new_error_num = current_node->label_false_num;
    new_label = 1;
}
else {
    new_error_num = current_node->label_true_num;
    new_label = -1;
}
int current_error_num = Error_num_cnt(current_node);
int current_leaves_num = Leaves_num_cnt(current_node);
double new_error = (double(root_error_num - current_error_num + new_error_num)
    + double(root_leaves_num - current_leaves_num + 1) * double(alpha)) / double(Train_row);

//计算剪枝前的融入模型复杂度的错误率
double current_error = (double(root_error_num) + double(root_leaves_num) * double(alpha)) / double(Train_row);

//判断错误率是否下降
if (new_error < current_error) {
    //生成叶子节点
    DecisionTree_node* sub_node = new DecisionTree_node();
    sub_node->property = -1;//定义-1为叶子节点
    sub_node->property_value = new_label;//根据多数投票的结果标签值进行定义
    sub_node->error_num = new_error_num;//记录多数投票结果的错误数
    //删除该节点原先的所有子节点决策树
    for (vector<DecisionTree_node*>::iterator i = current_node->child.begin(); i != current_node->child.end(); i++) {
        FreeDecisionTree(*i);
    }
    current_node->child.clear();
    current_node->child.push_back(sub_node);
    //更新决策树叶子节点总数、决策树错误数
    root_error_num = root_error_num - current_error_num + new_error_num;
    root_leaves_num = root_leaves_num - current_leaves_num + 1;
}
```

4.3. 可调叉数的多叉树模型

由于通过观察数据集发现，特征第 0 列和特征第 3 列的特征值取值范围偏广，这便会造成决策树某一节点的分支太多，而其他各列特征的特征值取值范围仅在 0-4 之间或 0-1 之间，所以我们希望将特征第 0 列和特征第 3 列的特征值的范围也能限定在一定小范围内，从而实现减少叉数的操作。

实现的办法很简单，我们直接对最开始的输入进来的训练集数据或测试集数据进行处理，通过取余操作便可以将一个大范围的连续的离散值映射到一个较小的范围上，而且也不会影响后期的建树，实现方便。

```
/*
特征值映射函数
输入：特征、特征值
输出：映射后的特征值
*/
int MapClass(int property, int current_value) {
    int branch = 4;//将分支个数限定为branch
    if (property == 0 || property == 3) {
        return int(current_value % branch);
    }
    else return current_value;
}
```

4.4. 随机森林（可结合前面的算法流程图）

首先读取训练集进行处理，然后通过随机采样一定数量的数据集及特征集进行建树，接着存储当前的决策树，重复上述操作得到随机森林，接着便能将此随机森林运用于测试集数据的决策得到标签值，最后通过多数投票的办法输出标签值。

4.4.1. 随机采样函数：通过随机采样一定数量的数据集及特征集

```
/*
训练集数据随机抽取
输入：训练集数据
输出：随机抽取的训练集数据
*/
vector<vector<int>> Random_data(vector<vector<int>> current_data) {
    int exist[1000] = { 0, };
    vector<vector<int>> sub_data;
    //随机抽取训练集数据
    for (int i = 0; i < 200;) {
        int j = rand() % current_data.size();
        if (exist[j] == 0) {
            sub_data.push_back(current_data[j]);
            i++;
            exist[j] = 1;
        }
    }
    return sub_data;
}
```

```
/*
特征集数据随机抽取
输入：特征集
输出：随机抽取的特征集
*/
vector<int> Random_property(vector<int> current_property) {
    int exist[10] = { 0, };
    vector<int> sub_property;
    //随机抽取特征集
    for (int i = 0; i < 6;) {
        int j = rand() % current_property.size();
        if (exist[j] == 0) {
            sub_property.push_back(current_property[j]);
            i++;
            exist[j] = 1;
        }
    }
    return sub_property;
}
```

4.5. 交叉验证法

由于这次实验只给了训练集没有验证集，所以自己去查看了交叉验证法的相应资料并加以实现，基本思路便是先将数据集随机打乱，然后将数据集分为 K 份（在这次实验中分了 8 份），然后交叉验证每次取 K-1 份数据作训练集，1 份作验证集，循环遍历 K 次，得到的 K 个准确率再取平均便得到了最终的准确率。由于代码过于简单只贴训练集读取的代码。

```
//读取训练集数据文件
ifstream train_file("train.csv");
int tem_row = 0;
if (train_file) {
    string train_str;
    while (getline(train_file, train_str)) {
        //根据交叉验证的分割份数读取文件
        if (!((file_split * 100) <= tem_row && tem_row < ((file_split + 1) * 100))) {
            //记录每行数据的特征值与标签值
            char* train_c_str = (char*)train_str.c_str();
            const char* d = ",";//以空格作截取词
            char* tem_c = strtok(train_c_str, d);//截取有效单词
            vector<int> tem_row_data;
            int tem_col = 0;
            while (tem_c) {
                //存储数据特征值,判断是否进行减少分叉操作
                if(ReduceClass == true) tem_row_data.push_back(MapClass(tem_col,atoi(tem_c)));
                else tem_row_data.push_back(atoi(tem_c));
                tem_c = strtok(NULL, d);
                tem_col++;
            }
            Train_data.push_back(tem_row_data);
            Train_row++;
        }
        tem_row++;
    }
    Train_col = Train_data[0].size();
    train_file.close();
}
```

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）

1.1. 训练集为

	A	B	C	D
1	1	1	1	1
2	1	1	6	1
3	1	0	5	-1
4	0	0	4	1
5	0	1	7	-1
6	0	1	3	-1
7	0	0	8	-1
8	1	0	7	1
9	0	1	5	-1

1.2. ID3 模型：

数据集 D 的经验熵: $-\frac{4}{9}\log_2\frac{4}{9}-\frac{5}{9}\log_2\frac{5}{9}=0.991bit$

特征 A 对数据集 D 的条件熵: $\frac{4}{9}\left(-\frac{3}{4}\log_2\frac{3}{4}-\frac{1}{4}\log_2\frac{1}{4}\right)+\frac{5}{9}\left(-\frac{1}{5}\log_2\frac{1}{5}-\frac{4}{5}\log_2\frac{4}{5}\right)=0.762bit$

特征 A 对数据集 D 的信息增益: $0.991-0.762=0.229bit$

特征 B 对数据集 D 的条件熵: $\frac{5}{9}\left(-\frac{2}{5}\log_2\frac{2}{5}-\frac{3}{5}\log_2\frac{3}{5}\right)+\frac{4}{9}(-\frac{2}{4}\log_2\frac{2}{4}-\frac{2}{4}\log_2\frac{2}{4})=0.984bit$

特征 B 对数据集 D 的信息增益: $0.991-0.984=0.007bit$

特征 C 对数据集 D 的条件熵: $0.222bit$

特征 C 对数据集 D 的信息增益: $0.991-0.222=0.769bit$

1.3. C4.5 模型：

数据集 D 对特征 A 的值的熵: $0.991bit$

特征 A 对数据集 D 的信息增益率: $(0.991-0.762)/0.991=0.232bit$

数据集 D 对特征 B 的值的熵: $0.991bit$

特征 B 对数据集 D 的信息增益率: $(0.991-0.984)/0.991=0.007bit$

数据集 D 对特征 C 的值的熵: $2.725bit$

特征 C 对数据集 D 的信息增益率: $(0.991-0.222)/2.725=0.282bit$

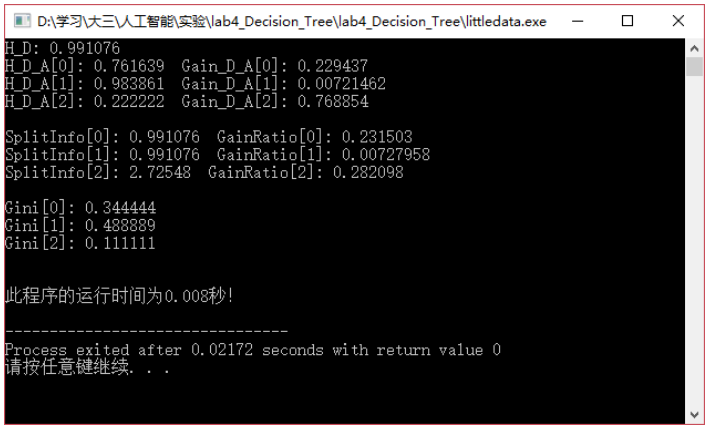
1.4. CART 模型：

特征 A 的条件下数据集 D 的 GINI 系数: $0.344bit$

特征 B 的条件下数据集 D 的 GINI 系数: $0.489bit$

特征 C 的条件下数据集 D 的 GINI 系数: $0.111bit$

1.5. 程序运行结果符合计算结果与预测结果，程序正确执行。



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）

2.1. 最优结果（交叉验证法）

决策模型	准确率
决策树（后剪枝、C4.5）	0.64375

2.2. 优化结果分析（交叉验证法）

我们可以看到本次实验优化前后的准确率并没有太大的变化，唯一产生可观变化的调整叉数的优化，但结果却随着叉数越小结果越差，**初步分析是此次数据集过少且数据集本身也是比较易分**，我们用最原始的办法建立决策树可以观察到，在有九个特征的情况下数据集所建立的决策树深度也就在 4、5 层左右就产生了叶子节点了，所以我们后续的优化操作，如剪枝对决策树的深度改变没有很明显的作用，也就导致了最后的结果没有我们想象中的明显提高。

注：由于这次实验数据集被随机打乱，且随机森林也是概率事件，所以下面展示数据具有一定随机性。而且由于优化较多，展示时使用控制变量法，即只改变某优化看结果，所以只有同一个优化分析中的结果有可比性，在不同的优化分析中由于没有很好控制变量，没有可比性。

2.2.1. 基于最普通的决策树（无任何优化）测试三种决策模型

决策模型	准确率
ID3	0.63
C4.5	0.6325
CART	0.63

2.2.2. 基于 C4.5 决策模型展示剪枝优化结果

优化前	准确率	优化后	准确率
普通决策树	0.6325	预剪枝	0.63625
		后剪枝	0.64375

2.2.3. 基于 C4.5 决策模型展示调整叉数优化结果

优化前	准确率	优化后	准确率
普通决策树	0.6325	二叉树	0.5275
		四叉树	0.55
		八叉树	0.62125

2.2.4. 基于 C4.5 决策模型展示随机森林优化结果

优化前	准确率	优化后	准确率
普通决策树	0.6325	随机森林	0.635

四、 思考题

1. 决策树有哪些避免过拟合的方法？

1.1. 剪枝（融于模型复杂度）：

- 预剪枝：在决策树生成过程中对于当前节点计算划分前后的决策树在验证集上的准确率，如果准确率不提高，则无需划分。若当前节点的深度已达限定深度，则无需划分，则将当前节点标记为叶节点，类别为父节点中出现最多的类。
- 后剪枝：生成完整的决策树后用后序遍历对于每个非叶节点，加入将它变成叶子节点，决策树在验证集上的准确率不降低，则将它变成叶子节点。

1.2. 随机森林：用随机的方式建立一个森林，森林里面有很多的决策树组成，随机森林的每一棵决策树之间是没有关联的，训练时通过随机对行与列进行采样得到数据进行建树。在得到森林之后，当有一个新的输入样本进入的时候，就让森林中的每一棵决策树分别进行一下判断，看看这个样本应该属于哪一类，然后看看哪一类被选择最多，就预测这个样本为那一类。

1.3. 使用不同的决策策略： 使用 C4.5 模型（信息增益率）或 CART 模型（GINI 系数）代替 ID3 模型，避免某个属性有较多分支使得训练得到一个庞大且深度浅的树。

2. C4.5相比于ID3的优点是什么？

2.1. ID3 模型是使用信息增益作为划分决策，信息增益越大，则选取该分裂规则。多分叉树。信息增益可以理解为，有了 x 以后对于标签 p 的不确定性的减少，减少的越多越好，即信息增益越大越好。但这也造成了 ID3 的缺点，倾向于选择水平数量较多的变量，可能导致训练得到一个庞大且深度浅的树；另外输入变量必须是分类变量（连续变量必须离散化）；最后无法处理空值。

2.2. C4.5模型选择了信息增益率替代信息增益。使用信息增益率作为划分决策（需要用信息增益除以，该属性本身的熵），因为ID3算法会偏向于选择类别较多的属性（形成分支较多会导致信息增益大），所以C4.5模型可以避免ID3算法中的归纳偏置问题，避免过拟合现象，比较分裂前后信息增益率，选取信息增益率最大的作为决策划分。

3. 如何用决策树来判断特征的重要性？

3.1. 对于一个新的数据集，将其输入到决策树中进行决策，首先从根节点开始进行决策判断，让当前节点的特征与数据集中的特征进行匹配，根据数据集的特征值不同，决策判断前往当前节点的不同分支，到达子节点再继续进行决策过程，直到遍历到叶子结点得到标签为止。而这一遍历过程中数据集依次遍历特征的顺序则决定了特征集中的特征对于当前数据集的重要性，越先遍历到的特征则越重要。