决策树算法的实现及其对于入噪增殖数据的分类研究

摘要

本文意在初步研究如今利用最广泛的数据挖掘算法之一–决策树算法。结合课设题目所给的天气分类的题目，首先实现了 *ID3* 算法。以天气状况最为第一分类点，当天晴时再以湿度分类，下雨时再以温度分类，多云是直接判定 *Y*。同时，为了解决 *ID3* 算法对于噪声数据处理困难，以及偏向较多数值的缺点，又实现了以信息增益比为分类标准的 *C4.5* 算法。当然，为了产生有噪声数据，针对原有数据集离散的特点，我建立了基于马尔科夫的增值模型，根据原有数据的概率分布，构建一步转移矩阵，产生新的数据集用来测试 *C4.5* 算法。

关键字：*ID3*；*C4.5*；状态转移矩阵；数据增殖；马尔科夫链

0 引言

数据分类是数据挖掘中一个重要的内容。分类存在很多方法，常见的分类模型有决策树、神经网络、遗传算法、粗糙集、统计模型等。其中决策树算法是以实例为基础的归纳学习算法，以其易于提取显示规则、计算量相对较小、可以显示重要决策属性和较高的分类准确率等优点而得到广泛的应用。据统计，目前决策树算法是利用最广泛的数据挖掘算法之一。

从改进 ID3 算法的角度:1998 年，刘小虎博士和李生教授 f191 提出，决策树优化是决策树学习算法中十分重要的分支。以 ID3 为基础，提出改进的递归信息增益优化算法。每当选择一个新的属性时，算法不仅仅是考虑该属性带来的信息增益，还需要考虑到该属性后选择的属性带来的信息增益，即同时考虑树的两层节点。2001 年，郭茂祖博士和刘扬教授针对 ID3 多值偏向的缺陷，提出了一种新的基于属性一值对为内节点的决策树归纳算法 AVPI，它所产生的决策树大小及测试速度均优于 ID3。

从构造机制的角度:1999 年，吴菲和黄梯云教授提出利用二元决策树实现模型选择，并采用遗传算法构造二元决策树并提出了遗传算法基于二元决策树的模型选择方法，以趋势预测模型为例进行了验证，获得了较好的效果。2005 年，黄沛等提出一种基于遗传算法的多重决策树组合分类方法，该算法与单个决策树相比，具有更高的分类精度。

从多变量模糊决策树的角度:2005 年，黄定轩等提出一类加权连续属性的多变量决策树构造方法，首先利用粗糙集理论和模糊聚类理论确定连续多变量属性的选择问题，然后利用聚类中心算法建立等级标准中心以解决连续变量的区间划分问题，其次将等价关系相对泛化的概念用于决策树中多变量检验的构造。2006 年，张曙红教授等给出了一种面向连续值属性的模糊粗糙集决策树分析方法。该方法基于模糊聚类对连续属性进行离散化，并通过计算模糊隶属度矩阵中条件属性和类属性之间的模糊依赖性量度来确定属性的重要性和发现冗余属性。

从新的决策树构造方法的角度:2003 年，杨宏伟博士和王熙照教授等用基于层次分解的方法通过产生多层决策树来处理多类问题。2006，阳东升博士等通过对组织协作网与决策树的描述分析提出了组织结构设计的新思路—基于决策个体在任务上的协作关系设计最佳的决策树。

1　问题重述-决策树算法的实现（编号 48）

简介：决策树是通过一系列规则对数据进行分类的过程。它提供一种在什么条件下会得到什么值的类似规则的方法。它是一个从上到下、分而治之的归纳过程，是决策树的一个经典的构造算法。应用于很多预测的领域，如通过对信用卡客户数据构建分类模型，可预测下一个客户他是否属于优质客户。

分类过程：分类是数据挖掘、机器学习和模式识别中一个重要的研究领域。数据分类是一个两步过程。第一步，使用已知类别标记的训练数据集建立一个分类模型。例如：图 1 是一个决策树模型。第二步，对未知标记的数据使用模型进行分类。例如，根据图 1 的决策树模型，运用自顶而下的属性测试过程，将表 2 中的样例 1-6 分别分类为“Y”、“Y”、“Y”、“Y”、“N”、“N”。

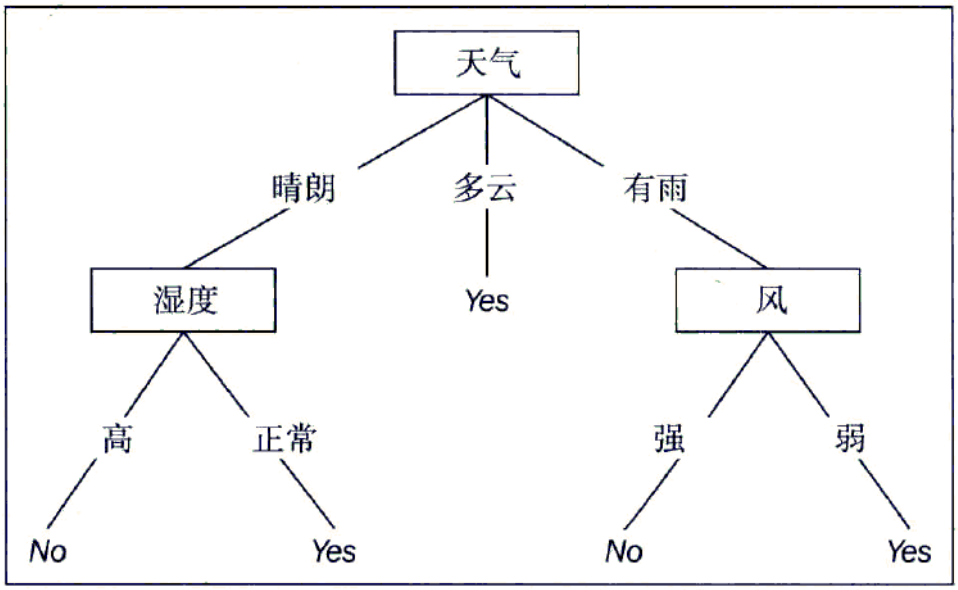


图 1　一个决策树模型的例子

算法描述：

输入：训练样例集 S，未标记的节点 T，属性集 A

输出：以 T 为根的决策树

□ 如果 S 中所有样例都是正例，则标记节点 T 为“Y”，并结束；

□ 如果 S 中所有样例都是反例，则标记节点 T 为“N”，并结束；

□ 否则，从 A 中选择一个属性 X，(可随机选) 标记节点 T 为 X;

□ 设 X 的所有取值为 V1, V2,…,Vn，依据这些取值将 S 划分为 n 个子集 S1, S2, …, Sn，建 T 的 n 个孩子节点 Ti，并分别以 Vi 作为从 T 到 Ti 的分支标号;

□ 对每对（Si，Ti，A-X），递归调用 ID3 算法建立一棵以 Ti 为根的子树；END

举例：对下表 1 运用算法构建决策树

表 1: 天气分类样本数据集

编号	天况	温度	湿度	风况	分类
1	晴	热	大	无	N
2	晴	热	大	有	N
3	多云	热	大	无	Y
4	雨	中	大	无	Y
5	雨	冷	正常	无	Y
6	雨	冷	正常	有	N
7	多云	冷	正常	有	Y
8	晴	中	大	无	N
9	晴	冷	正常	无	Y
10	雨	中	正常	无	Y
11	晴	中	正常	有	Y
12	多云	中	大	有	Y
13	多云	热	正常	无	Y
14	雨	中	大	有	N

对下列样例输入使用构建的决策树模型预测其分类属性：

表 2: 待分类数据集

编号	天况	温度	湿度	风况	分类
1	晴	热	正常	无	?
2	晴	热	正常	有	?
3	雨	热	正常	无	?
4	晴	中	正常	无	?
5	晴	冷	大	有	?
6	晴	冷	大	无	?

要求：

□ 设计合理的数据结构，编程实现决策树构造算法；

□ 给定训练数据集，运用构建的决策树模型，设计合理的文件格式，保存于外存之中；

□ 设计决策树分类算法，根据保存在外存的决策树模型，实现决策树的分类过程，完成对未知类别属性数据样例的分类。

2 基于 ID3 的决策树模型与求解

ID3 算法是决策树的一种，它是基于奥卡姆剃刀原理的，即用尽量用较少的东西做更多的事。ID3 算法，即 Iterative Dichotomiser 3，迭代二叉树 3 代，是 Ross Quinlan 发明的一种决策树算法，这个算法的基础就是上面提到的奥卡姆剃刀原理，越是小型的决策树越优于大的决策树，尽管如此，也不总是生成最小的树型结构，而是一个启发式算法。

2.1 信息熵与信息增益

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。ID3 算法的核心思想就是以信息增益来度量属性的选择，选择分裂后信息增益最大的属性进行分裂。该算法采用自顶向下的贪婪搜索遍历可能的决策空间。

在信息增益中，重要性的衡量标准就是看特征能够为分类系统带来多少信息，带来的信息越多，该特征越重要。在认识信息增益之前，先来看看信息熵的定义。

熵这个概念最早起源于物理学，在物理学中是用来度量一个热力学系统的无序程度，而在信息学里面，熵是对不确定性的度量。在 1948 年，香农引入了信息熵，将其定义为离散随机事件出现的概率，一个系统越是有序，信息熵就越低，反之一个系统越是混乱，它的信息熵就越高。所以信息熵可以被认为是系统有序化程度的一个度量。

假如一个随机变量 X 的取值为 $X = \{x_1, x_2, ..., x_n\}$，每一种取到的概率分别是 $\{p_1, p_2, ..., p_n\}$，那么 X 的熵定义为:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{1}$$

意义为一个变量的变化情况可能越多，那么它携带的信息量就越大。

对于分类系统来说，类别是变量，它的取值是，而每一个类别出现的概率分别是 $\{P_{C_1}, P_{C_2}, ..., P_{C_n}\}$ 而这里的就是类别的总数，此时分类系统的熵就可以表示为:

$$H(C) = -\sum_{i=1}^{n} P_{C_i} \log_2 P_{C_i} \tag{2}$$

以天气分类为例，如上表一所示是描述天气的数据表：可以看出，一共 14 个样例，包括 9 个正例和 5 个负例。那么当前信息的熵计算如下：

$$Entropy(S) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940286 \tag{3}$$

在决策树分类问题中，信息增益就是决策树在进行属性选择划分前和划分后信息的差值。假设利用属性‘天况’来分类，那么如下图：

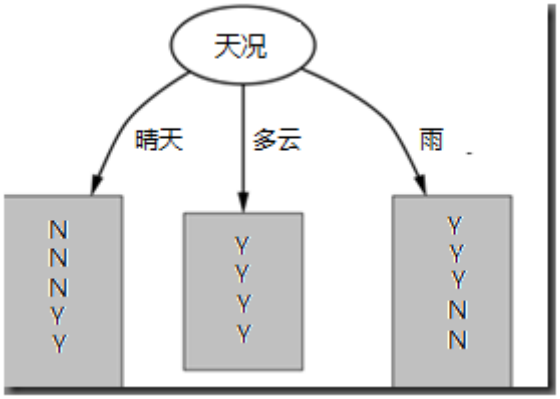


图 2　天况分类图

划分后，数据被分为三部分了，那么各个分支的信息熵计算如下:

$$Entropy(S) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.970951 \tag{4}$$

$$Entropy(S) = -\frac{4}{4}\log_2\frac{4}{4} - 0 * \log_2 0 = 0 \tag{5}$$

$$Entropy(S) = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0.970951 \tag{6}$$

那么划分后的信息熵为：

$$Entropy(S|T) = -\frac{5}{14} * 0.970951 + \frac{4}{14} * 0 + \frac{5}{14} * 0.970951 = 0.693536 \tag{7}$$

$Entropy(S|T)$ 代表在特征属性的条件下样本的条件熵。那么最终得到特征属性带来的信息增益为：

$$IG(T) = Entropy(S) - Entropy(S|T) = 0.24675 \tag{8}$$

信息增益的计算公式如下:

$$IG(S|T) = Entropy(S) - \sum_{value(T)} \frac{|S_v|}{S} Entropy(S_v) \tag{9}$$

其中 S 为全部样本集合，$value(T)$ 是属性 T 所有取值的集合，V 是 T 的其中一个属性值，S_v 是 S 中属性 T 的值为的样例集合，为中所含样例数。

为了通俗理解上述的过程。示意图如下 3,4,5.

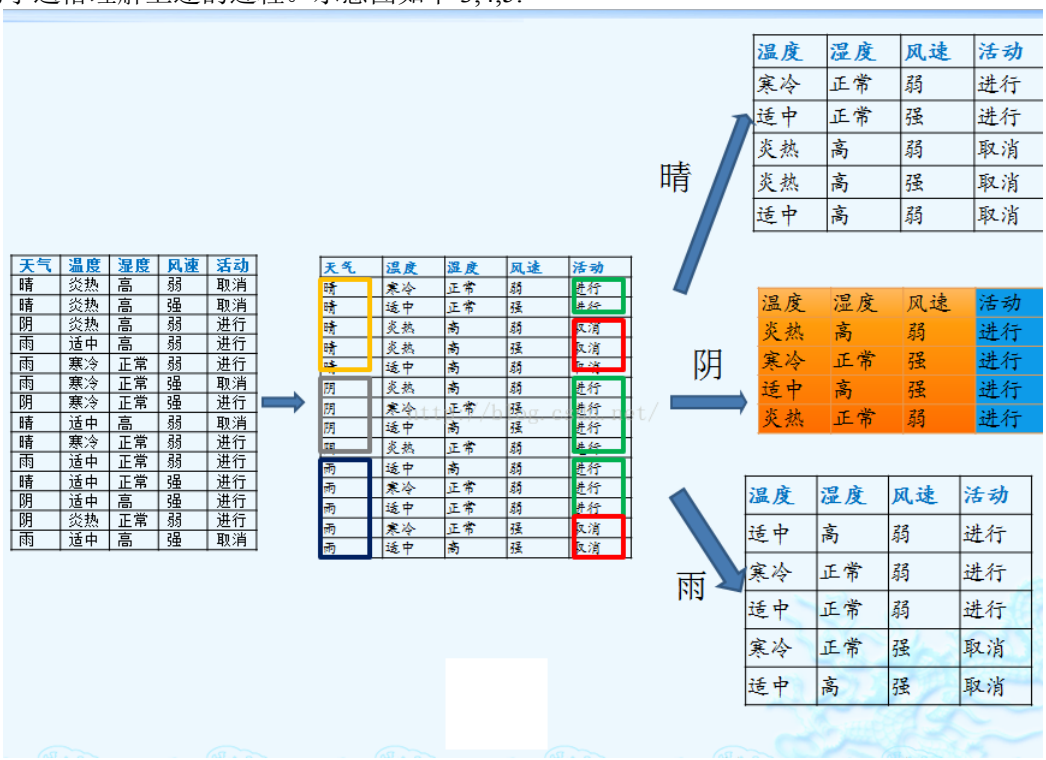


图 3

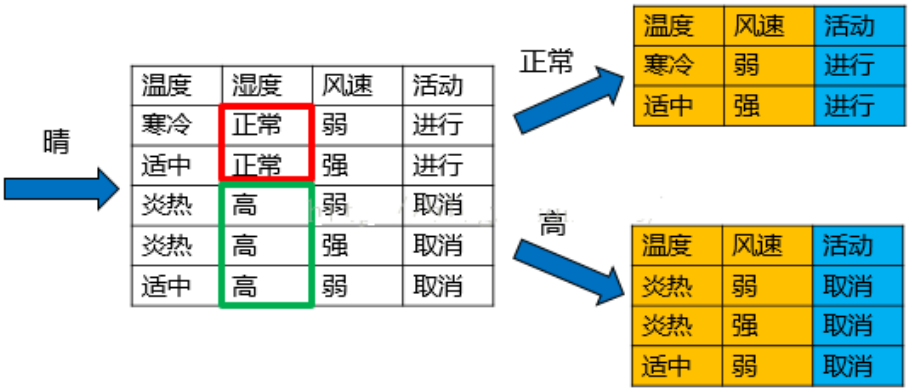


图 4

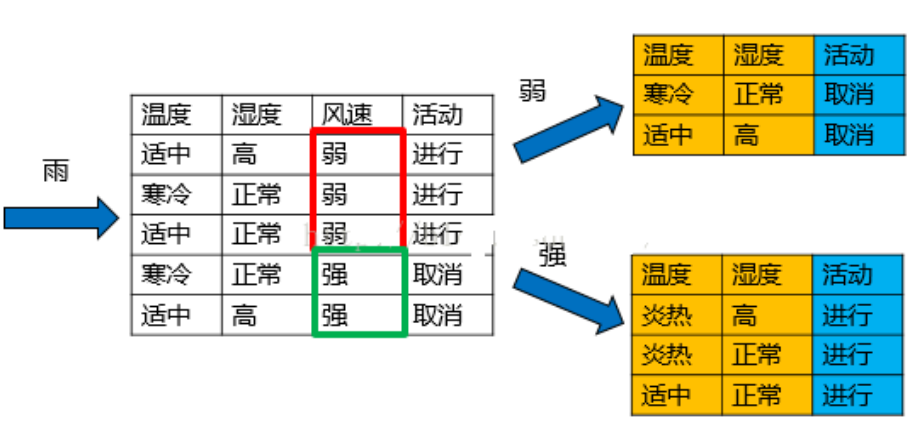


图 5

在决策树的每一个非叶子结点划分之前，先计算每一个属性所带来的信息增益，选择最大信息增益的属性来划分，因为信息增益越大，区分样本的能力就越强，越具有代表性，很显然这是一种自顶向下的贪心策略。以上就是 ID3 算法的核心思想。

2.2 ID3 算法的实现

伪代码：

输入：训练集 $D = \{(x_1, y_1), ((x_2, y_2), ..., ((x_m, y_m)\}$

属性集 $A = \{a_1, a_2, ..., a_d\}$。

过程：函数 TreeGenerate(D, A)

生成结点 node;

if D 中样本全属于同一类别 C then

将 node 标记为 C 类叶结点; return;

end if

if A == 空集（ORD 中样本在 A 上取值相同）then

将 node 标记为叶结点，其类别标记为 D 中样本数最多的类; return

end if

从 A 中选择最优划分属性 a*;

for a* 的每一个值 a*（vdo）

为 node 生成一个分支; 令 D_v 表示 D 中在 a* 上取值为 $a*_v$ 的样本子集;

if D_v 为空 then

将分支结点标记为叶结点，其类别标记为 D 中样本最多的类;then

else

以 TreeGenerte(D_v, A a*) 为分支结点

end if

end for

输出：以 node 为根节点的一颗决策树。

流程图如下：

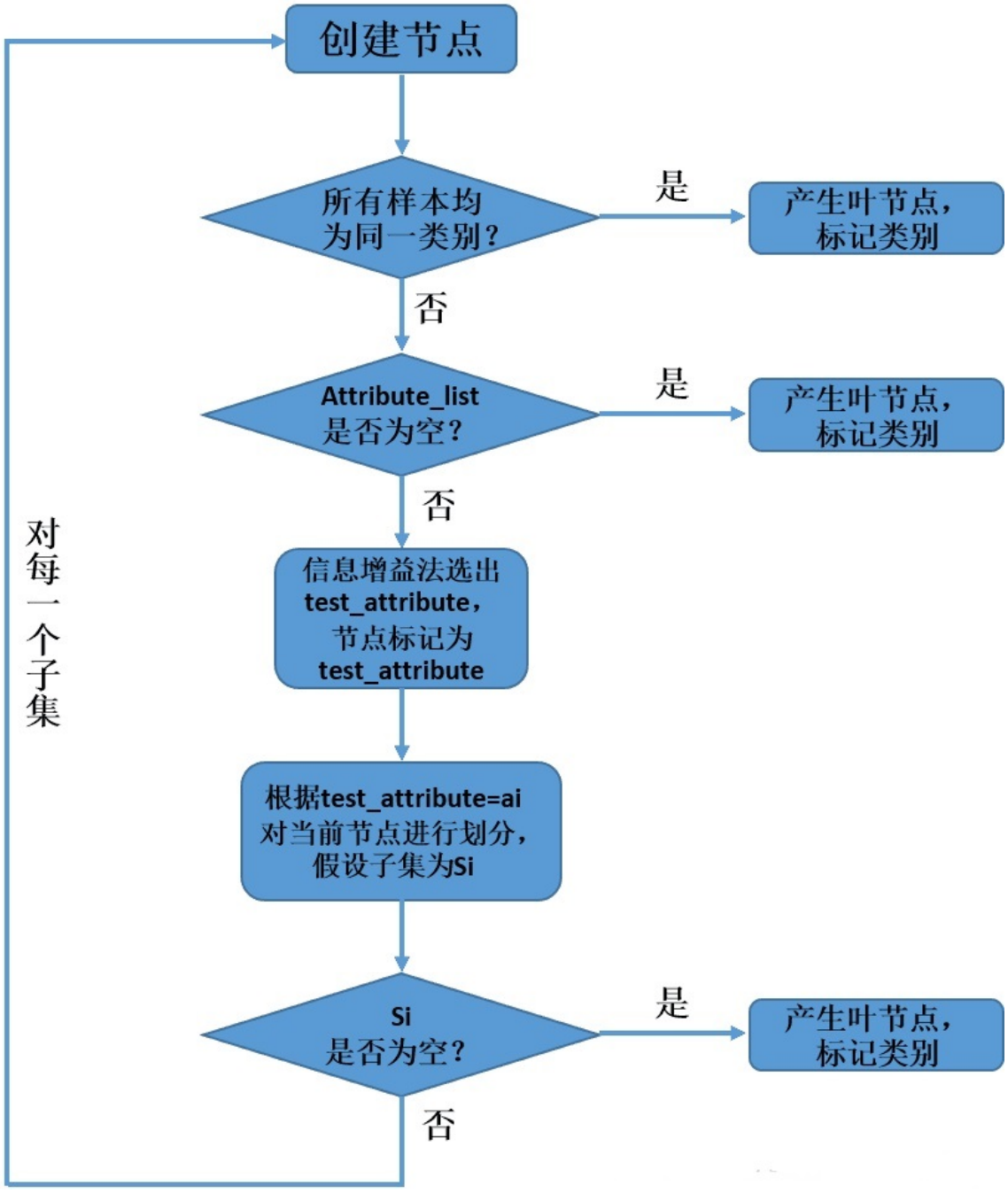


图 6　算法流程图

2.3　题目实例详解

在 2.1 中我们以第一步分类为例详解了如何利用属性天气时如何计算信息增益。在本小节中，

将详细介绍整个分类过程。

同 2.1 的原理计算出其他 3 个非类别属性的信息增益，取最大的那个属性作为分裂节点，此例中最大的是 Outlook，进而得到如下图 4 所示：

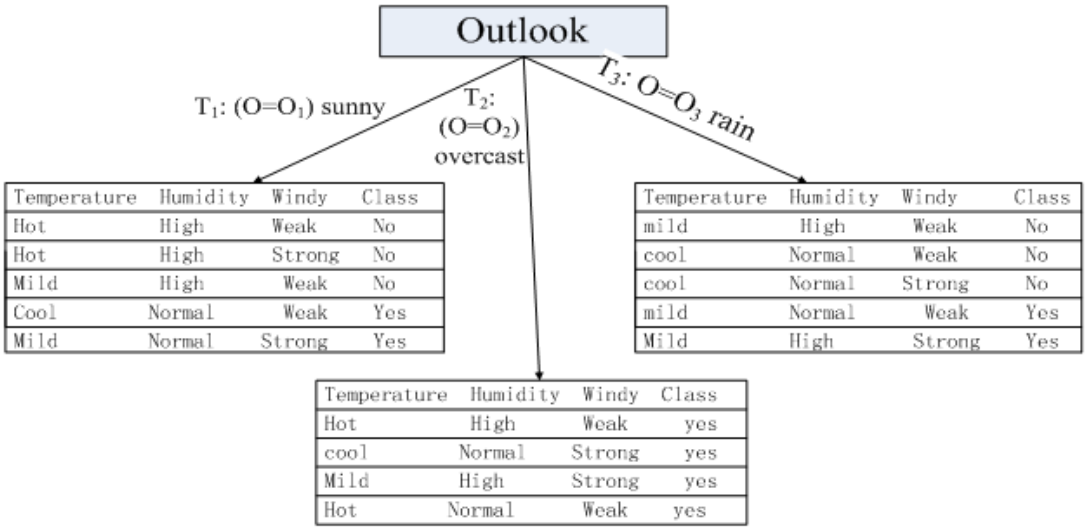


图 7

上图中，针对 sunny 中的子训练数据集分支，有两个类别，该分支中有 3 个实例属于 no 类，有 2 个实例属于 yes 类，求类别属性新的信息熵

$$I_{T_i}(p,n) = -\frac{2}{5}log_2\frac{2}{5} - \frac{3}{5}log_2\frac{3}{5} \tag{10}$$

再分别求 3 个非类别属性的信息熵，同时求出各属性的信息增益，选出信息增益最大的属性 Humidity，得：

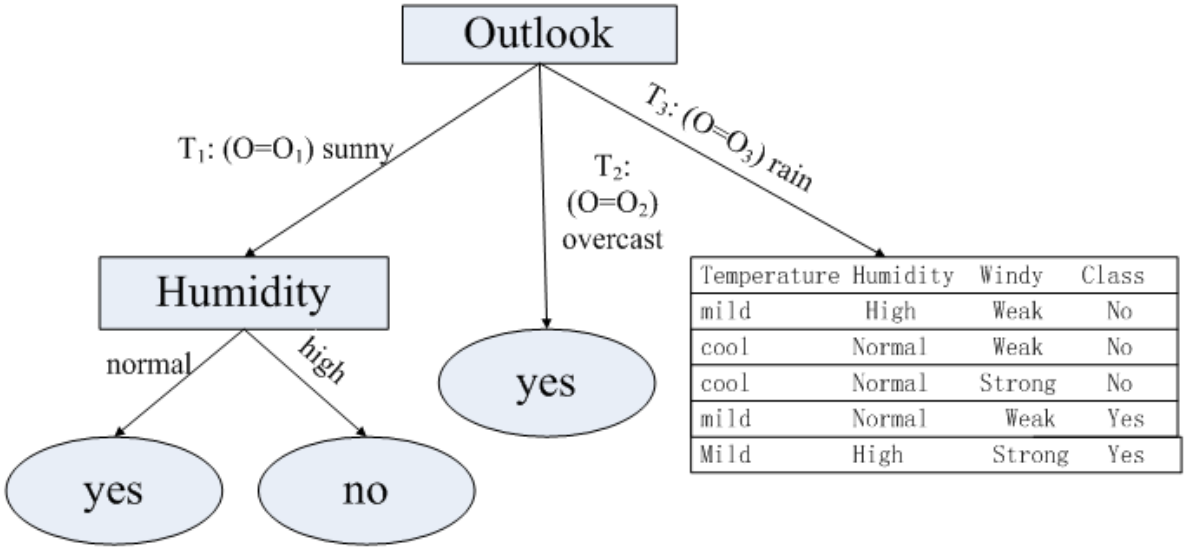


图 8

同理可得：

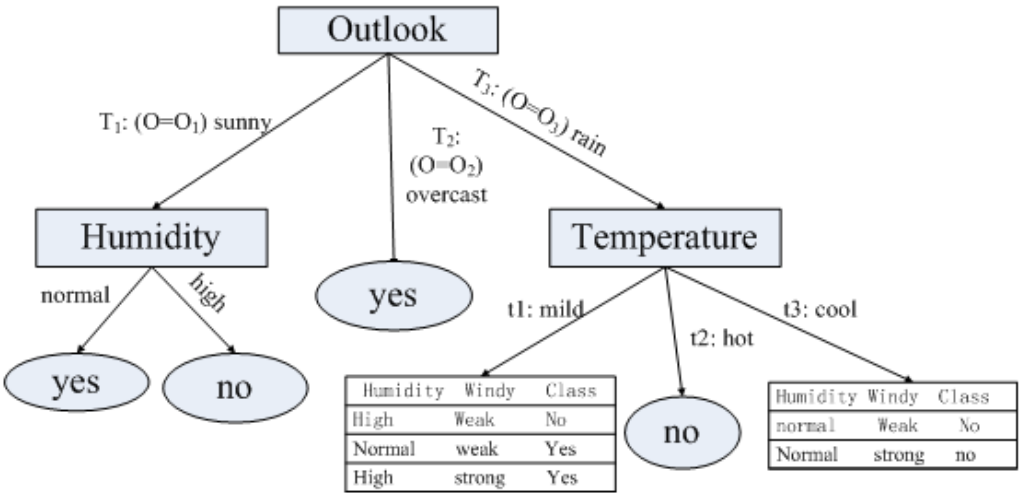


图 9

cool 对应的数据子集都是 no，所以直接写 no，无需分裂。mild 对应的数据子集，Humidity 和 windy 的信息增益是相同的，因为在该分组中，yes 元组的比例比 no 元组的大，所以直接写 yes，最终得到的决策树图如图所示：

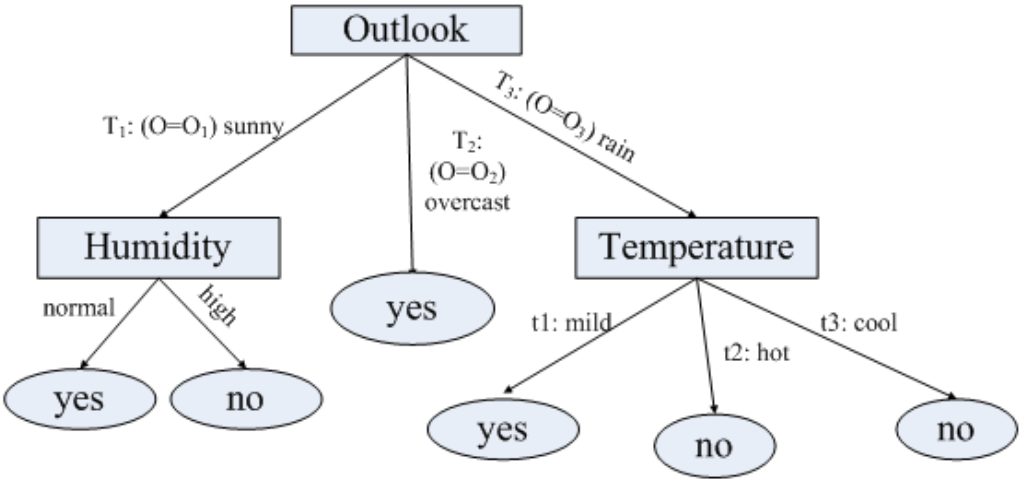


图 10

包含信息的详细决策树为：

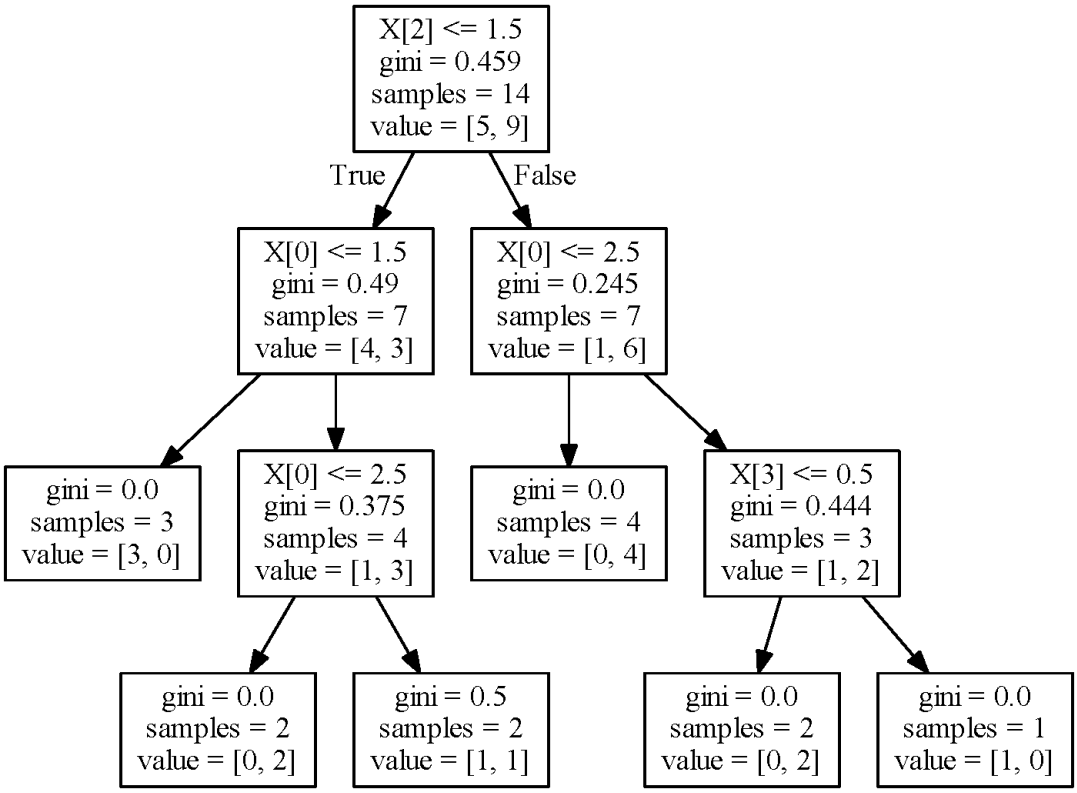


图 11

3 模型优化与改进

想要对第二节中的模型进行优化与改进，就要先了解 ID3 算法的优缺点。从目前的资料中很容易了解到：

优点：

1. 决策树易于理解和解释，可以可视化分析。

2. 决策树分类器的构造不需要任何领域知识或参数设置，

3. 适合高维数据

4. 可以同时处理标称型和数值型数据

5. 计算复杂度不高

缺点：

1. 容易出现过拟合

2. 对缺失数据处理比较困难

3. 忽略数据集中属性的相关性

4.ID3 算法计算信息增益时偏向数值较多的特征

5. 不支持增量学习

因为时间有限，本文，针对缺点的 2 和 4 做出优化，建立基于 C4.5 算法的改进决策树模型。

3.1 C4.5 算法的实现

ID3 使用信息增益作为特征选择的度量，而 C4.5 使用信息增益比作为特征选择的度量。

使用信息增益的话其实是有一个缺点，那就是它偏向于具有大量值的属性。什么意思呢？就是说在训练集中，某个属性所取的不同值的个数越多，那么越有可能拿它来作为分裂属性。例如一个训练集中有 10 个元组，对于某一个属相 A，它分别取 1-10 这十个数，如果对 A 进行分裂将会分成 10 个类，那么对于每一个类 $Info(D_j)$=0，从而式为 0，该属性划分所得到的信息增益最大，但是很显然，这种划分没有意义。

正是基于此，ID3 后面的 C4.5 采用了信息增益率这样一个概念。信息增益率使用“分裂信息”值将信息增益规范化。分类信息类似于 Info(D)，定义如下

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} * log_2(\frac{|D_j|}{|D|}) \tag{11}$$

这个值表示通过将训练数据集 D 划分成对应于属性 A 测试的 v 个输出的 v 个划分产生的信息。信息增益率定义：

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo(A)} \tag{12}$$

选择具有最大增益率的属性作为分裂属性。

3.2 基于马尔科夫的数据集增殖

为了检验 3.1 模型中 C4.5 对于有噪声数据的处理效果，我建立了基于马尔科夫的数据集增殖模型, 对于原有数据进行增殖建模，以得到更多数据集。　　在给定的数据集上，每一天的状态判断，可以看做是一个离散的序列记为 $\{X_n = X(N), n = 1, 2, ...\}$, 它可以看做在时间集 $T_1 = 0, 1, 2, ...$ 上对离散的马氏过程相继观察的结果。我们约定记链的状态空间 $I = a_1, a_{2,..}$ 为在链的状态下，马尔科夫性通常用条件分布律来表示，即满足

$$everyn \in Z, r \in Z, 0 <= t_1 < t_2 < ... < t_r < m; t_i, m, n + m \in T_1 \tag{13}$$

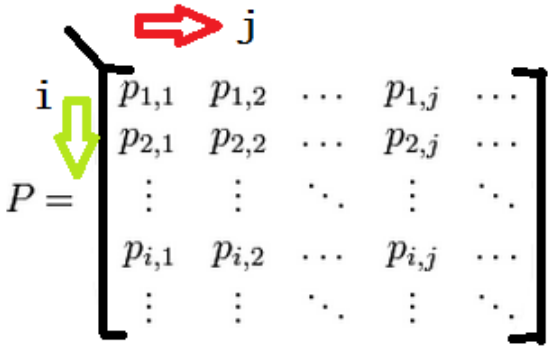
有

$$PX_{m+n} = a_j | X_{t1} = a_{t1}, X_{t2} = a_{t2}, ..., X_{tr} = a_{tr}, X_m = a_t = PX_{m+n} = a_j | X_m = a_i \qquad a_i \in I \tag{14}$$

记右段上式为 $P_{ij}(m, m + n)$ 称条件概率

$$P_{ij}(m, m + n) = PX_{m+n} = a_j | X_m = a_j \tag{15}$$

为马氏链在时刻 m 处于状态 a_j 条件下，在时刻 m+n 转移到 a_j 的转移概率。在本课设中我们我们只考虑一步转移概率矩阵。如下：



$p_{i,j} = Pr(j|i)$ 表示由i转变到j的概率，是以在i为条件的。

核心代码部分如下：

```
count = {}
for i in x[0:len(x)-1]:
    count[i] = count.get(i, 0) + 1
count = sorted(count.items(), key=op.itemgetter(0), reverse=False)

markov_marix = np.zeros([len(count), len(count)])
for j in range(len(x)-1):
    for m in range(len(count)):
        for n in range(len(count)):
            if x[j] == count[m][0] and x[j+1] == count[n][0]:
                markov_marix[m][n] += 1
for t in range(len(count)):
    markov_marix[t, :] /= count[t][1]
print(markov_marix)
```

executed in 135ms, finished 01:16:40 2019-05-08

```
[[0.   0.   0.   0.4  0.4  0.2 ]
 [0.   0.   0.   0.5  0.25 0.25]
 [0.   0.   0.   0.   0.6  0.4 ]
 [0.25 0.25 0.5  0.   0.   0.  ]
 [0.4  0.4  0.2  0.   0.   0.  ]
 [0.25 0.25 0.5  0.   0.   0.  ]]
```

可以随机生成一组数据。如下图 12

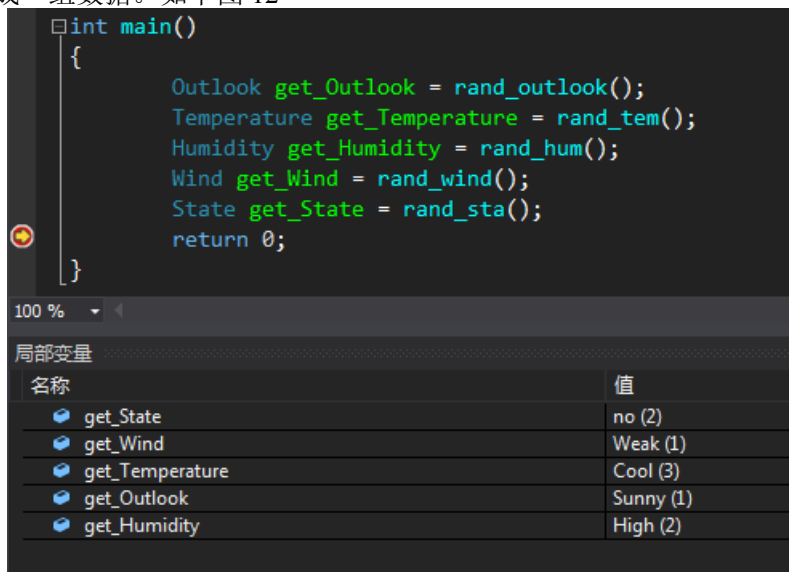


图 12

4 实验与检验

ID3 算法构建决策树如下图 13

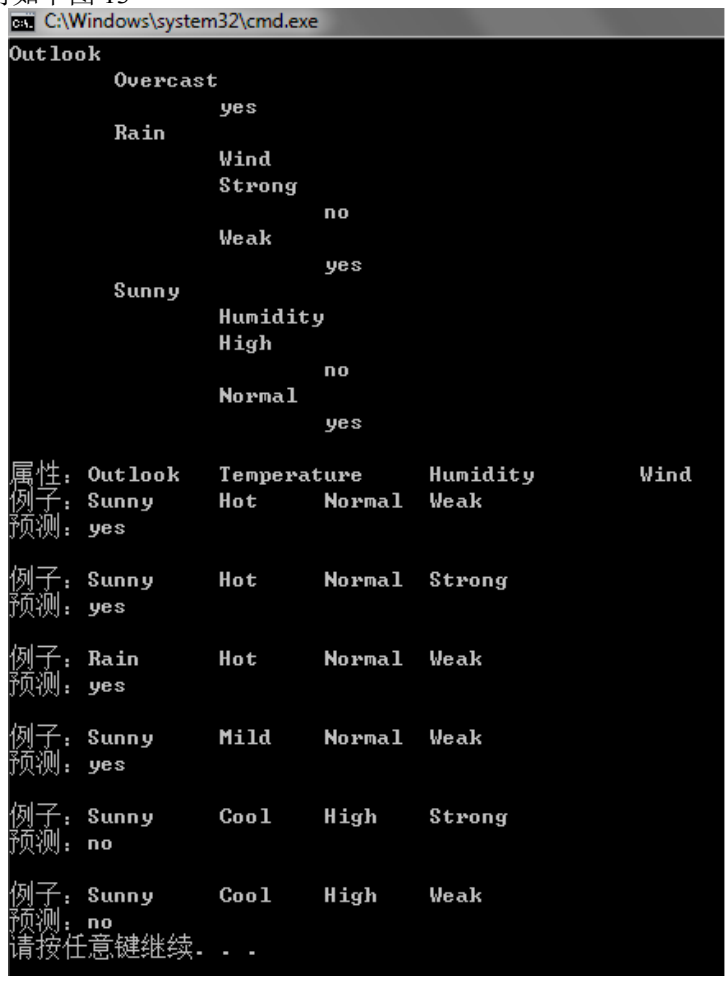


图 14　ID3 构建决策树

状态转移矩阵为

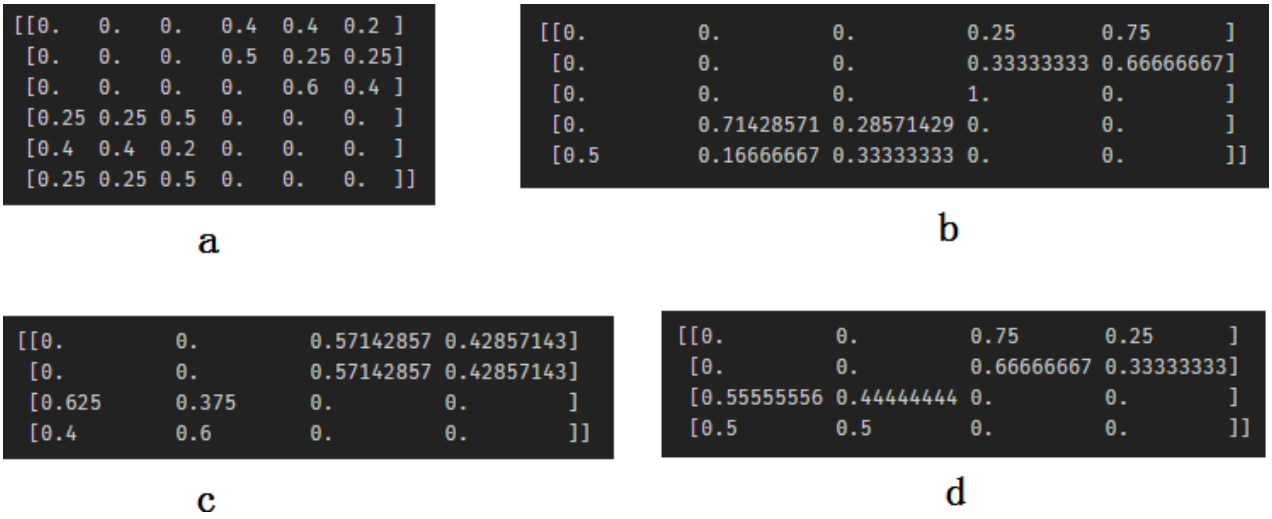


图 16　状态转移矩阵

C4.5 算法实现

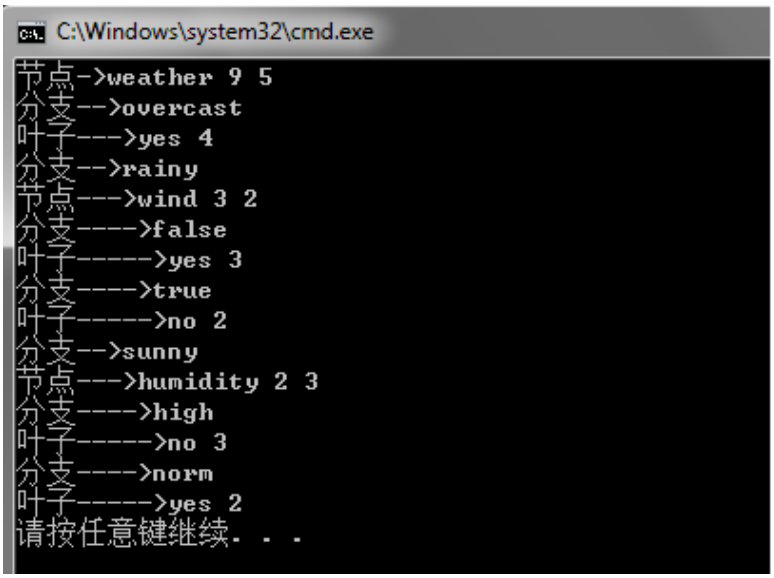


图 17　C4.5 算法实现

5　源代码

5.1　ID3 算法实现

运行环境：win7+vs2013

```
#include "stdafx.h"
#include <iostream>
```

函数名称	类型	功能	参数	描述
createDataset	void	创建数据集	X	将 x 的值放入 X 中
			attributes	压入具体属性
calcShanno	double	计算给定数据集的香农熵	data	要计算数据集的信息熵，有效信息为 yes 和 no
			classCounts	统计 data 中 yes 和 no 的数量
splitDataSet	vector<vector<string>>	按照给定特征划分数据集，划分后的数据集中不包含给定特征，即新的数据集的维度少了一个	data	需要划分特征的数据集
			axis	特征下标(0,1,2,3)
			value	特征值(string:每种属性的具体种类)
createFeatureList chooseBestFeatureToSplit	vector<string> int	创建特征列表 选择最好的数据集划分方式	data	要提取特征列表的数据集
			axis	需要提取哪个属性的特征
			featureList	返回某一属性的所有特征
			data	要选择划分方式的数据集
			bestFeature	当前分类的最好属性
			bestInfoGain	最大的信息增益
			subData	是数据集除去 featureList[j] 之后的数据集
			newEntropy	一次累加得到除去该特征之后的香农熵
createTree	Node*	递归构建决策树	root	当前根节点
			data	要分类的数据集
			attribute	属性列表
			classList	收集当前数据的所有的 yes no 到 classList
			classList1	所有 yes no 的种类的
			bestFeatureIndex	这里得到当前情况下以哪种属性分类
			subAttribute	新的属性列表，不包含当前要划分的特征，即消耗了一个特征，特征的维度少了一个

```cpp
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <map>
#include <math.h>
using namespace std;

/*set的特性是，所有元素都会根据元素的键值自动排序，set的元素不像map那样可以同时拥有实值(value)和键值(key),set元素的键值就是实值，实值就是键值。set不允许两个元素有相同的键值*/
/*算法（Algorithm）为一个计算的具体步骤，常用于计算、数据处理和自动推理。C++ 算法库（Algorithms library）为 C++ 程序提供了大量可以用来对容器及其它序列进行算法操作的函数。这些组件可以为函数或函数模板，大部份由头文件 <algorithm> 提供，一小部份位于 <numeric>、<cstdlib> 中。*/
#define N 14
#define feature 4

vector< vector<string> > X;
string x[N][feature + 1] =
{
{ "Sunny", "Hot", "High", "Weak", "no" },
{ "Sunny", "Hot", "High", "Strong", "no" },
{ "Overcast", "Hot", "High", "Weak", "yes" },
{ "Rain", "Mild", "High", "Weak", "yes" },
{ "Rain", "Cool", "Normal", "Weak", "yes" },
{ "Rain", "Cool", "Normal", "Strong", "no" },
{ "Overcast", "Cool", "Normal", "Strong", "yes" },
{ "Sunny", "Mild", "High", "Weak", "no" },
{ "Sunny", "Cool", "Normal", "Weak", "yes" },
{ "Rain", "Mild", "Normal", "Weak", "yes" },
{ "Sunny", "Mild", "Normal", "Strong", "yes" },
{ "Overcast", "Mild", "High", "Strong", "yes" },
{ "Overcast", "Hot", "Normal", "Weak", "yes" },
{ "Rain", "Mild", "High", "Strong", "no" },
};
```

```
34
35  //属性的名称，比如 Outlook取值：Sunny，Overcast，Rain
36  string attribute[] =   {"Outlook","Temperature","Humidity","Wind
        "};
37
38  vector<string> attributes;
39
40  /*
41  ********************************************************************

42  函数名称：createDataset
43  功能：创建数据集
44  参数：无
45  返回值：无（操作对象为全局变量）
46  ********************************************************************

47  */
48
49  void createDataset()//创建数据集
50  {
51  //创建数据集//N*(feature + 1)的二维vector
52  X = vector< vector<string> >(N, vector<string>(feature + 1));
53  //将x的值放入X中
54  int i, j;
55  for (i = 0; i<N; i++)
56  {
57  for (int j = 0; j<feature + 1; j++)
58  {
59  X[i][j] = x[i][j];
60  }
61  }
62
63  //创建属性
64  for (i = 0; i<feature; i++)//这里用vector的push_back函数，从尾部
        插入数字
65  attributes.push_back(attribute[i]);
66  }
```

```
67
68  /*
69  ****************************************************************
70  函数名称：calcShanno
71  功能：计算给定数据集的香农熵
72  参数：string二维容器（引用）
73  返回值：给定数据集的香农熵（double）
74  ****************************************************************
75  */
76
77  double calcShanno(vector< vector<string> > &data)//计算给定数据
        集的香农熵
78  {
79  int n = data.size(); //二维容器一共多少行
80  map<string, int> classCounts; //构架键实对
81  int i;
82  int label = data[0].size() - 1; //标签的数量为每行数量－1（方便
        classCounts下标对应最后的分类，其实就是让下标对应yes no）
83  for (i = 0; i<n; i++)  //初始为0//每一行的yes no对应的实值都初始
        化为0
84  {
85  classCounts[data[i][label]] = 0; //关键字都是“yes”“no”，只不
        过对应的实值都是0
86  }
87  for (i = 0; i<data.size(); i++)  //每当出现一次，+1
88  {
89  classCounts[data[i][label]] += 1; //这里累计yes,no的次数//这里就
        体现了map键对应的值可以被修改的属性
90  }
91
92  //计算香农熵
93  double shanno = 0;
94  map<string, int>::iterator it; //迭代器是指针的泛指
95  for (it = classCounts.begin(); it != classCounts.end(); it++)//
        这里一共就两个键值对yes－9,no－5(当时这是第一次的时候)
```

```
96  {
97  double prob = (double)(it->second) / (double)n;//最后类别所占比
        例//这个对的second就是yes或者no的数量
98  shanno -= prob * (log(prob) / log(2));
99  }
100 return shanno;
101 }
102
103 /*
104 ************************************************************************

105 函数名称: splitDataSet
106 功能: 按照给定特征划分数据集, 划分后的数据集中不包含给定特征, 即
        新的数据集的维度少了一个
107 参数: string二维容器, axis : 特征下标(0,1,2,3), value: 特征值(
        string:每种特征的具体种类)
108 返回值: 给定数据集的香农熵(double)
109 ************************************************************************

110 */
111
112 vector< vector<string> > splitDataSet(vector< vector<string> >
        data, int axis, string value)//按照给定特征划分数据集, 划分后
        的数据集中不包含给定特征, 即新的数据集的维度少了一个
113 {
114 vector< vector<string> > result;
115 for (int i = 0; i<data.size(); i++)//data.size=14
116 {
117 if (data[i][axis] == value)//data[i].size=5
118 {
119 //将"当前特征"这个维度去掉
120 vector<string> removed(data[i].begin(), data[i].begin() + axis);
        //初始化为两个迭代器指定范围中元素的拷贝
121 removed.insert(removed.end(), data[i].begin() + axis + 1, data[i
        ].end());  //v.insert(v.end(), a[1], a[3]);//在尾部插入a[1]个
        a[3]//当然这里是指两个指针的的数据
122 result.push_back(removed);
```

```
123 //其实这里不是去掉，而是重新定义了数据集result，然后把除去<axis，
        value>这个，剩下的复制进去而已
124 }
125 }
126 return result;
127 }
128
129 /*
130 ************************************************************************

131 函数名称：createFeatureList
132 功能：创建特征列表
133 参数：string二维容器（引用），  axis ：特征下标(0,1,2,3)
134 返回值：特征的所有取值（string）
135 ************************************************************************

136 */
137
138 vector<string> createFeatureList(vector< vector<string> > &data,
        int axis)//创建特征列表
139 {
140 int n = data.size();            //=14
141 vector<string>featureList;    //某个属性的特征的所有取值（如
        outlook的Sunny，Rain，overcast）
142 set<string> s;
143 for (int j = 0; j<n; j++)     //寻找该特征的所有可能取值
144 s.insert(data[j][axis]); //把该特征所有取值都插入s容器
145 set<string>::iterator it;
146 for (it = s.begin(); it != s.end(); it++)
147 {
148 featureList.push_back(*it);//再把s容器所有值都插入特征列表
149 }
150 return featureList;
151 }
152
153 /*
154 ************************************************************************
```

```
155 函数名称：chooseBestFeatureToSplit
156 功能：选择最好的数据集划分方式
157 参数：string二维容器（引用）
158 返回值：当前分类的最好属性
159 ************************************************************

160 */
161
162 int chooseBestFeatureToSplit(vector< vector<string> > &data)//选
    择最好的数据集划分方式
163 {
164 int n = data[0].size() - 1;                                    //=4
165 double bestEntropy = calcShanno(data);  //初始香农熵
166 double bestInfoGain = 0;                                  //最大的信息增益
167 int bestFeature = 0;                                          //最好的
    属性
168 for (int i = 0; i<n; i++)                                     //所有特
    征
169 {
170 double newEntropy = 0;
                                //初始化新的香农熵
171 vector<string> featureList = createFeatureList(data, i);  //该特
    征的所有可能取值（从0、1、2、3，即所有特征对应的值）//其实这
    里相当于某个特征的那一列。
172 for (int j = 0; j<featureList.size(); j++)
                    //featureList.size()=14
173 {
174 vector< vector<string> > subData = splitDataSet(data, i,
    featureList[j]); //这里的subData是数据集除去featureList[j]之
    后的数据集
175 double prob = (double)subData.size() / (double)data.size(); //除
    去当前特征之后，剩余数据集所占比例
176 newEntropy += prob * calcShanno(subData); //一次累加得到除去该特
    征之后的香农熵
177 }
178 double infoGain = bestEntropy - newEntropy;  //信息增益，即熵的
```

```
    减少，或数据无序度的减少
179 if (infoGain > bestInfoGain) //这里是选取当前情况下，哪种特征的
    信息增益最大
180 {
181 bestInfoGain = infoGain;
182 bestFeature = i;
183 }
184 }
185 return bestFeature; //这里返回的是标号0、1、2、3
186 }
187
188 /*
189 ************************************************************************

190 函数名称：majorityCnt
191 功能：返回出现次数最多的分类名称
192 //如果数据集已处理了所有属性，但类标签依然不是唯一的，采用多数表
    决的方法定义叶子节点的分类
193 参数：string容器（引用）
194 返回值：string
195 ************************************************************************

196 */
197
198 string majorityCnt(vector<string> &classList)
199 {
200 int n = classList.size();
201 map<string, int> classCount;
202 int i;
203 for (i = 0; i<n; i++)
204 {
205 classCount[classList[i]] = 0;
206 }
207 for (i = 0; i<n; i++)
208 {
209 classCount[classList[i]] += 1;
210 }
```

```
211
212 int maxCnt = 0;
213 map<string, int>::iterator it;
214 string result = "";
215 for (it = classCount.begin(); it != classCount.end(); it++)
216 {
217 if (it->second > maxCnt)
218 {
219 maxCnt = it->second;
220 result = it->first;
221 }
222 }
223 return result;
224 }
225
226 struct Node //子叶节点
227 {
228 string attribute; //属性
229 string val; //特征值(string:每种特征的具体种类)
230 bool isLeaf; //是否有孩子结点
231 vector<Node*> childs; //孩子节点指针
232
233 Node() //构造函数
234 {
235 val = "";
236 attribute = "";
237 isLeaf = false;
238 }
239 };
240
241 Node *root = NULL; //静态结点
242
243 /*
244 ************************************************************************

245 函数名称: createTree
246 功能: 递归构建决策树
```

```
247 参数: Node*根节点,二维容器string（引用）, string容器: 属性
248 返回值: Note*结点地址
249 ****************************************************************
250 */
251
252 Node* createTree(Node *root, vector< vector<string> > &data,
        vector<string> &attribute)
253 {
254 if (root == NULL)
255 root = new Node();
256 vector<string> classList;
257 set<string> classList1; //这里运用了set关键字不可重复的特点, 来
        作为检查样本数据是否有yes或no
258 int i, j;
259 int label = data[0].size() - 1; //=4
260 int n = data.size(); //=14
261 for (i = 0; i<n; i++) //收集当前数据的所有的yes no到classList,
        所有yes no的种类的到classList1
262 {
263 classList.push_back(data[i][label]);
264 classList1.insert(data[i][label]);
265 }
266 if (classList1.size() == 1)  //如果所有实例都属于同一类, 停止划
        分（只有yes或者no）
267 {
268 if (classList[0] == "yes")
269 root->attribute = "yes";
270 else
271 root->attribute = "no";
272 root->isLeaf = true;
273 return root;
274 }
275 if (data[0].size() == 1)  //遍历完所有特征, 返回出现次数最多的类
        别
276 {
277 root->attribute = majorityCnt(classList);
```

```
278 return root;
279 }
280
281 int bestFeatureIndex = chooseBestFeatureToSplit(data); //这里得
    到当前情况下以哪种属性分类
282 vector<string> featureList = createFeatureList(data,
    bestFeatureIndex); //从上面得到的当前分类的最佳属性分类，然
    后这里把属性的所有可能特征值都放到链表
283 string bestFeature = attribute[bestFeatureIndex]; //这里就体现了
    bestFeatureIndex = chooseBestFeatureToSplit(data)是int型的
284
285 root->attribute = bestFeature;    //记录要划分的特征，放入结点
286
287 for (i = 0; i<featureList.size(); i++)  //对于当前属性的每个可能
    值，创建新的分支
288 {
289 vector<string> subAttribute;  //新的属性列表，不包含当前要划分的
    特征，即消耗了一个特征，特征的维度少了一个
290 for (j = 0; j<attribute.size(); j++)
291 {
292 if (bestFeature != attribute[j])
293 subAttribute.push_back(attribute[j]); //把，不包含当前要划分的属
    性，的所有特征值全部压入容器
294 }
295 Node *newNode = new Node();
296 newNode->val = featureList[i];    //记录属性的取值（取哪个特征字
    符串）
297 createTree(newNode, splitDataSet(data, bestFeatureIndex,
    featureList[i]), subAttribute);//此时用函数splitDataSet创建一
    个新的数据集，当然这里排除了bestFeatureIndex，同时，把剩余的
    属性放进去
298 root->childs.push_back(newNode); //然后把结点压入孩子指针容器
299 }
300 return root;
301 }
302
303 /*
```

```
304 ************************************************************************
305 函数名称: print
306 功能: 打印
307 参数: Node*根节点,树的深度
308 返回值: 无
309 ************************************************************************
310 */
311
312 void print(Node *root, int depth)
313 {
314 int i;
315 for (i = 0; i<depth; i++)
316 cout << "\t";
317
318 if (root->val != "")
319 {
320 cout << root->val << endl;
321 for (i = 0; i<depth + 1; i++)
322 cout << "\t";
323 }
324 cout << root->attribute << endl;
325 vector<Node*>::iterator it;
326 for (it = root->childs.begin(); it != root->childs.end(); it++)
327 {
328 print(*it, depth + 1);
329 }
330 }
331
332 /*
333 ************************************************************************
334 函数名称: clsssify
335 功能: 预测
336 参数: Node*根节点,属性列表(string容器),
337 返回值: 无
```

```
338 ************************************************************************
339 */
340
341 string classify(Node *root, vector<string> &attribute, string *
        test)
342 {
343 string firstFeature = root->attribute;
344 int firstFeatureIndex;
345 int i;
346 for (i = 0; i<feature; i++)  //找到根节点是第几个特征
347 {
348 if (firstFeature == attribute[i])
349 {
350 firstFeatureIndex = i;
351 break;
352 }
353 }
354 if (root->isLeaf)  //如果是叶子节点，直接输出结果
355 return root->attribute;
356 for (i = 0; i<root->childs.size(); i++)
357 {
358 if (test[firstFeatureIndex] == root->childs[i]->val)
359 {
360 return classify(root->childs[i], attribute, test);
361 }
362 }
363 }
364
365 //释放节点
366 void freeNode(Node *root)
367 {
368 if (root == NULL)
369 return;
370 vector<Node*>::iterator it;
371 for (it = root->childs.begin(); it != root->childs.end(); it++)
372 freeNode(*it);
```

```
373 delete root;
374 }
375
376 int main(void)
377 {
378 createDataset();
379 root = createTree(root, X, attributes);
380 print(root, 0);
381
382 string test[6][4] =
383 {{"Sunny","Hot","Normal","Weak"},
384 {"Sunny","Hot","Normal","Strong"},
385 {"Rain","Hot","Normal","Weak"},
386 {"Sunny","Mild","Normal","Weak"},
387 {"Sunny","Cool","High","Strong"},
388 {"Sunny","Cool","High","Weak"}};
389 int i,j;
390 cout << endl << "属性: ";
391 for (i = 0; i<feature; i++)
392 cout << attributes[i] << "\t";
393
394
395 for (j = 0; j < 6; j++)
396 {
397 cout << endl << "例子: ";
398 for (i = 0; i < feature; i++)
399 {
400 cout << test[j][i] << "\t";
401 }
402 cout << endl << "预测: ";
403 cout << classify(root, attributes, test[j]) << endl;
404 }
405
406 freeNode(root);
407
408 return 0;
409 }
```

5.2 状态转移矩阵

运行环境：Anaconda3+Jupyter notebook

```
import numpy as np
import operator as op
from enum import Enum
class outlook(Enum):
Sunny = 1
Overcast = 2
Rain = 3

class temperature(Enum):
Hot = 4
Mild = 5
Cool = 6
count = {}
for i in x[0:len(x)-1]:
count[i] = count.get(i, 0) + 1
count = sorted(count.items(), key=op.itemgetter(0), reverse=
    False)

markov_marix = np.zeros([len(count), len(count)])
for j in range(len(x)-1):
for m in range(len(count)):
for n in range(len(count)):
if x[j] == count[m][0] and x[j+1] == count[n][0]:
markov_marix[m][n] += 1
for t in range(len(count)):
markov_marix[t, :] /= count[t][1]
print(markov_marix)
```

```
import numpy as np
import operator as op
from enum import Enum

```

```
5  class temperature(Enum):
6  Hot = 4
7  Mild = 5
8  Cool = 6
9
10 class humidity(Enum):
11 Normal=7
12 High=8
13
14 x = np.array([temperature.Hot.value,
15 humidity.High.value,
16 temperature.Hot.value,
17 humidity.High.value,
18 temperature.Hot.value,
19 humidity.High.value,
20 temperature.Mild.value,
21 humidity.High.value,
22 temperature.Cool.value,
23 humidity.Normal.value,
24 temperature.Cool.value,
25 humidity.Normal.value,
26 temperature.Cool.value,
27 humidity.Normal.value,
28 temperature.Mild.value,
29 humidity.High.value,
30 temperature.Cool.value,
31 humidity.Normal.value,
32 temperature.Mild.value,
33 humidity.Normal.value,
34 temperature.Mild.value,
35 humidity.Normal.value,
36 temperature.Mild.value,
37 humidity.High.value,
38 temperature.Hot.value,
39 humidity.Normal.value,
40 temperature.Mild.value,
41 humidity.High.value
```

```
])

count = {}
for i in x[0:len(x)-1]:
count[i] = count.get(i, 0) + 1
count = sorted(count.items(), key=op.itemgetter(0), reverse=
    False)

markov_marix = np.zeros([len(count), len(count)])
for j in range(len(x)-1):
for m in range(len(count)):
for n in range(len(count)):
if x[j] == count[m][0] and x[j+1] == count[n][0]:
markov_marix[m][n] += 1
for t in range(len(count)):
markov_marix[t, :] /= count[t][1]
print(markov_marix)
```

```
import numpy as np
import operator as op
from enum import Enum

class humidity(Enum):
Normal=7
High=8

class wind(Enum):
Weak=9
Strong=10

x = np.array([humidity.High.value,
wind.Weak.value,
humidity.High.value,
wind.Strong.value,
humidity.High.value,
wind.Weak.value,
humidity.High.value,
```

```
20  wind.Weak.value,
21  humidity.Normal.value,
22  wind.Weak.value,
23  humidity.Normal.value,
24  wind.Strong.value,
25  humidity.Normal.value,
26  wind.Strong.value,
27  humidity.High.value,
28  wind.Weak.value,
29  humidity.Normal.value,
30  wind.Weak.value,
31  humidity.Normal.value,
32  wind.Weak.value,
33  humidity.Normal.value,
34  wind.Strong.value,
35  humidity.High.value,
36  wind.Strong.value,
37  humidity.Normal.value,
38  wind.Weak.value,
39  humidity.High.value,
40  wind.Strong.value
41  ])
42
43  count = {}
44  for i in x[0:len(x)-1]:
45  count[i] = count.get(i, 0) + 1
46  count = sorted(count.items(), key=op.itemgetter(0), reverse=
        False)
47
48  markov_marix = np.zeros([len(count), len(count)])
49  for j in range(len(x)-1):
50  for m in range(len(count)):
51  for n in range(len(count)):
52  if x[j] == count[m][0] and x[j+1] == count[n][0]:
53  markov_marix[m][n] += 1
54  for t in range(len(count)):
55  markov_marix[t, :] /= count[t][1]
```

```
56 print(markov_marix)
```

```
1  import numpy as np
2  import operator as op
3  from enum import Enum
4
5  class wind(Enum):
6  Weak=9
7  Strong=10
8
9  class state(Enum):
10 yes=11
11 no=12
12 x = np.array([wind.Weak.value,
13 state.no.value,
14 wind.Strong.value,
15 state.no.value,
16 wind.Weak.value,
17 state.yes.value,
18 wind.Weak.value,
19 state.yes.value,
20 wind.Weak.value,
21 state.yes.value,
22 wind.Strong.value,
23 state.no.value,
24 wind.Strong.value,
25 state.yes.value,
26 wind.Weak.value,
27 state.no.value,
28 wind.Weak.value,
29 state.yes.value,
30 wind.Weak.value,
31 state.yes.value,
32 wind.Strong.value,
33 state.yes.value,
34 wind.Strong.value,
35 state.yes.value,
```

```
wind.Weak.value,
state.yes.value,
wind.Strong.value,
state.yes.value
])

count = {}
for i in x[0:len(x)-1]:
count[i] = count.get(i, 0) + 1
count = sorted(count.items(), key=op.itemgetter(0), reverse=
    False)

markov_marix = np.zeros([len(count), len(count)])
for j in range(len(x)-1):
for m in range(len(count)):
for n in range(len(count)):
if x[j] == count[m][0] and x[j+1] == count[n][0]:
markov_marix[m][n] += 1
for t in range(len(count)):
markov_marix[t, :] /= count[t][1]
print(markov_marix)
```

5.3 马尔科夫增值数据

运行环境：win7+vs2013

```
#include"stdafx.h"
#include <random>
#include <stdlib.h>
#include<cmath>
#include <time.h>
using namespace std;
enum Outlook {Sunny=1,Overcast=2,Rain=3};
enum Temperature {Hot=1,Mild=2,Cool=3};
enum Humidity {Normal=1,High=2};
enum Wind {Weak=1,Strong=2};
enum State{yes=1,no=2};

```

```
13 Outlook rand_outlook()
14 {
15 srand((unsigned)time(NULL));
16 int n = 3;
17 double t, p1, p2, p3;
18 Outlook get;
19 p1 = 5. / 14;
20 p2 = 4. / 14;
21 p3 = 5. / 14;
22 //随机生成一个n（如100）以内的数,
23 t = rand() % n + 1;
24 if (t < n *p1)
25 {
26 get = Sunny;
27 return get;
28 //拿到1
29 }
30 else if (t < n*(p1 + p2))
31 {
32 get = Overcast;
33 return get;
34 //拿到2
35 }
36 else if (t<n*(p1 + p2 + p3))
37 {
38 get = Rain;
39 return get;
40 //拿到3
41 }
42 else
43 {
44 get = Rain;
45 return get;
46 }
47 }
48
49 Temperature rand_tem()
```

```
{
srand((unsigned)time(NULL));
int n = 3;
double t, p1, p2, p3;
Outlook before = rand_outlook();
Temperature get;
if (before == Sunny)
{
p1 = .4;
p2 = .4;
p3 = .2;
t = rand() % n + 1;
if (t < n *p1)
{
get = Hot;
return get;
}
else if (t < n*(p1 + p2))
{
get = Mild;
return get;
}
else if (t<n*(p1 + p2 + p3))
{
get = Cool;
return get;
}
else
{
get = Cool;
return get;
}

}

else if (before == Overcast)
{
```

```
87  p1 = .5;
88  p2 = .25;
89  p3 = .25;
90  t = rand() % n + 1;
91  if (t < n *p1)
92  {
93  get = Hot;
94  return get;
95  }
96  else if (t < n*(p1 + p2))
97  {
98  get = Mild;
99  return get;
100 }
101 else if (t<n*(p1 + p2 + p3))
102 {
103 get = Cool;
104 return get;
105 }
106 else
107 {
108 get = Cool;
109 return get;
110 }
111
112 }
113
114 else
115 {
116 p1 = 0;
117 p2 = .6;
118 p3 = .4;
119 t = rand() % n + 1;
120 if (t < n *p1)
121 {
122 get = Hot;
123 return get;
```

```
124 }
125 else if (t < n*(p1 + p2))
126 {
127 get = Mild;
128 return get;
129 }
130 else if (t<n*(p1 + p2 + p3))
131 {
132 get = Cool;
133 return get;
134 }
135 else
136 {
137 get = Cool;
138 return get;
139 }
140 }
141
142 }
143
144
145 Humidity rand_hum()
146 {
147 srand((unsigned)time(NULL));
148 int n = 2;
149 double t, p1, p2;
150 Temperature before = rand_tem();
151 Humidity get;
152 if (before == Hot)
153 {
154 p1 = .25;
155 p2 = .75;
156 t = rand() % n + 1;
157 if (t < n *p1)
158 {
159 get = High;
160 return get;
```

```
161 }
162 else if (t < n*(p1 + p2))
163 {
164 get = Normal;
165 return get;
166 }
167 else
168 {
169 get = Normal;
170 return get;
171 }
172
173 }
174
175 else if (before == Mild)
176 {
177 p1 = 1./3;
178 p2 = 2./3;
179 t = rand() % n + 1;
180 if (t < n *p1)
181 {
182 get = High;
183 return get;
184 }
185 else if (t < n*(p1 + p2))
186 {
187 get = Normal;
188 return get;
189 }
190
191 else
192 {
193 get = Normal;
194 return get;
195 }
196
197 }
```

```
198
199 else
200 {
201 p1 = 1;
202 p2 = 0;
203 t = rand() % n + 1;
204 if (t < n *p1)
205 {
206 get = High;
207 return get;
208 }
209 else if (t < n*(p1 + p2))
210 {
211 get = Normal;
212 return get;
213 }
214 else
215 {
216 get = Normal;
217 return get;
218 }
219 }
220 }
221
222
223 Wind rand_wind()
224 {
225 srand((unsigned)time(NULL));
226 int n = 2;
227 double t, p1, p2;
228 Humidity before = rand_hum();
229 Wind get;
230 if (before == Normal)
231 {
232 p1 = 0.57142857;
233 p2 = 0.42857143;
234 t = rand() % n + 1;
```

```
235 if (t < n *p1)
236 {
237 get = Weak;
238 return get;
239 }
240 else if (t < n*(p1 + p2))
241 {
242 get = Strong;
243 return get;
244 }
245 else
246 {
247 get = Strong;
248 return get;
249 }
250
251 }
252
253 else
254 {
255 p1 = 0.57142857;
256 p2 = 0.42857143;
257 t = rand() % n + 1;
258 if (t < n *p1)
259 {
260 get = Weak;
261 return get;
262 }
263 else if (t < n*(p1 + p2))
264 {
265 get = Strong;
266 return get;
267 }
268 else
269 {
270 get = Strong;
271 return get;
```

```
272 }
273 }
274 }
275
276
277 State rand_sta()
278 {
279 srand((unsigned)time(NULL));
280 int n = 2;
281 double t, p1, p2;
282 Wind before = rand_wind();
283 State get;
284 if (before == Weak)
285 {
286 p1 = 2. / 3;
287 p2 = 1. / 3;
288 t = rand() % n + 1;
289 if (t < n *p1)
290 {
291 get = yes;
292 return get;
293 }
294 else if (t < n*(p1 + p2))
295 {
296 get = no;
297 return get;
298 }
299 else
300 {
301 get = no;
302 return get;
303 }
304
305 }
306
307 else
308 {
```

```
309 p1 = 2. / 3;
310 p2 = 1. / 3;
311 t = rand() % n + 1;
312 if (t < n *p1)
313 {
314 get = yes;
315 return get;
316 }
317 else if (t < n*(p1 + p2))
318 {
319 get = no;
320 return get;
321 }
322 else
323 {
324 get = no;
325 return get;
326 }
327 }
328 }
329
330 int main()
331 {
332 Outlook get_Outlook = rand_outlook();
333 Temperature get_Temperature = rand_tem();
334 Humidity get_Humidity = rand_hum();
335 Wind get_Wind = rand_wind();
336 State get_State = rand_sta();
337 return 0;
338 }
```

5.4 C4.5 决策树算法的实现

运行环境：win7+vs2013

```
1 //tryC4.5.cpp
2 #include "stdafx.h"
3 #include "DecisionTree.h"
```

```
4
5  int main(int argc, char* argv[]){
6  string filename = "source.txt";
7  DecisionTree dt;
8  int attr_node = 0;
9  TreeNode* treeHead = nullptr;
10 set<int> readLineNum;
11 vector<int> readClumNum;
12 int deep = 0;
13 if (dt.pretreatment(filename, readLineNum, readClumNum) == 0)
14 {
15 dt.CreatTree(treeHead, dt.getStatTree(), dt.getInfos(),
       readLineNum, readClumNum, deep);
16 }
17 return 0;
18 }
19 /*
20 * @function CreatTree 预处理函数, 负责读入数据, 并生成信息矩阵和
       属性标记
21 * @param: filename 文件名
22 * @param: readLineNum 可使用行set
23 * @param: readClumNum 可用属性set
24 * @return int 返回函数执行状态
25 */
26 int DecisionTree::pretreatment(string filename, set<int>&
       readLineNum, vector<int>& readClumNum)
27 {
28 ifstream read(filename.c_str());
29 string itemline = "";
30 getline(read, itemline);
31 istringstream iss(itemline);
32 string attr = "";
33 while (iss >> attr)
34 {
35 attributes* s_attr = new attributes();
36 s_attr->attriName = attr;
37 //初始化属性名
```

```
38  statTree.push_back(s_attr);
39  //初始化属性映射
40  attr_clum[attr] = attriNum;
41  attriNum++;
42  //初始化可用属性列
43  readClumNum.push_back(0);
44  s_attr = nullptr;
45  }
46
47  int i = 0;
48  //添加具体数据
49  while (true)
50  {
51  getline(read, itemline);
52  if (itemline == "" || itemline.length() <= 1)
53  {
54  break;
55  }
56  vector<string> infoline;
57  istringstream stream(itemline);
58  string item = "";
59  while (stream >> item)
60  {
61  infoline.push_back(item);
62  }
63
64  infos.push_back(infoline);
65  readLineNum.insert(i);
66  i++;
67  }
68  read.close();
69  return 0;
70  }
71
72  int DecisionTree::statister(vector<vector<string>>& infos,
        vector<attributes*>& statTree,
73  set<int>& readLine, vector<int>& readClumNum)
```

```
74  {
75  //yes的总行数
76  int deciNum = 0;
77  //统计每一行
78  set<int>::iterator iter_end = readLine.end();
79  for (set<int>::iterator line_iter = readLine.begin(); line_iter
        != iter_end; ++line_iter)
80  {
81  bool decisLine = false;
82  if (infos[*line_iter][attriNum - 1] == "yes")
83  {
84  decisLine = true;
85  deciNum++; //无用, 因为子树的和就是他, 在计算时自动就加完这个数
        了
86  }
87  //如果该列未被锁定并且为属性列, 进行统计
88  for (int i = 0; i < attriNum - 1; i++)
89  {
90  if (readClumNum[i] == 0)
91  {
92  std::string tempitem = infos[*line_iter][i];
93  auto map_iter = statTree[i]->attriItem.find(tempitem);
94  //没有找到
95  if (map_iter == (statTree[i]->attriItem).end())
96  {
97  //新建
98  attrItem* attritem = new attrItem();
99  attritem->itemNum.push_back(1);
100 decisLine ? attritem->itemNum.push_back(1) : attritem->itemNum.
        push_back(0);
101 attritem->itemLine.insert(*line_iter);
102 //建立属性名->item映射
103 (statTree[i]->attriItem)[tempitem] = attritem;
104 attritem = nullptr;
105 }
106 else
107 {
```

```
108 (map_iter->second)->itemNum[0]++;
109 (map_iter->second)->itemLine.insert(*line_iter);
110 if (decisLine)
111 {
112 (map_iter->second)->itemNum[1]++;
113 }
114 }
115 }
116 }
117 }
118 return deciNum;
119 }
120
121 /*
122 * @function CreatTree 递归DFS创建并输出决策树
123 * @param: treeHead 为生成的决定树
124 * @param: statTree 为状态树，此树动态更新，但是由于是DFS对数据更
      新，所以不必每次新建状态树
125 * @param: infos 数据信息
126 * @param: readLine 当前在infos中所要进行统计的行数，由函数外给出
127 * @param: deep 决定树的深度，用于打印
128 * @return void
129 */
130 void DecisionTree::CreatTree(TreeNode* treeHead, vector<
      attributes*>& statTree, vector<vector<string>>& infos,
131 set<int>& readLine, vector<int>& readClumNum, int deep)
132 {
133 //有可统计的行
134 if (readLine.size() != 0)
135 {
136 string treeLine = "";
137 for (int i = 0; i < deep; i++)
138 {
139 treeLine += "--";
140 }
141 //清空其他属性子树，进行递归
142 resetStatTree(statTree, readClumNum);
```

```
143 //统计当前readLine中的数据：包括统计哪几个属性、哪些行，
144 //并生成statTree（由于公用一个statTree，所有用引用代替），并返回
    目的信息数
145 int deciNum = statister(getInfos(), statTree, readLine,
    readClumNum);
146 int lineNum = readLine.size();
147 int attr_node = compuDecisiNote(statTree, deciNum, lineNum,
    readClumNum);//本条复制为局部变量
148 //该列被锁定
149 readClumNum[attr_node] = 1;
150 //建立树根
151 TreeNode* treeNote = new TreeNode();
152 treeNote->m_sAttribute = statTree[attr_node]->attriName;
153 treeNote->m_iDeciNum = deciNum;
154 treeNote->m_iUnDecinum = lineNum - deciNum;
155 if (treeHead == nullptr)
156 {
157 treeHead = treeNote; //树根
158 }
159 else
160 {
161 treeHead->m_vChildren.push_back(treeNote); //子节点
162 }
163 cout << "节点-" << treeLine << ">" << statTree[attr_node]->
    attriName << " " << deciNum << " " << lineNum - deciNum <<
    endl;
164 //从孩子分支进行递归
165 for (map<string, attrItem*>::iterator map_iterator = statTree[
    attr_node]->attriItem.begin();
166 map_iterator != statTree[attr_node]->attriItem.end(); ++
    map_iterator)
167 {
168 //打印分支
169 int sum = map_iterator->second->itemNum[0];
170 int deci_Num = map_iterator->second->itemNum[1];
171 cout << "分支——" << treeLine << ">" << map_iterator->first <<
    endl;
```

```
172 //递归计算、创建
173 if (deci_Num != 0 && sum != deci_Num)
174 {
175 //计算有效行数
176 set<int> newReadLineNum = map_iterator->second->itemLine;
177 //DFS
178 CreatTree(treeNote, statTree, infos, newReadLineNum, readClumNum
        , deep + 1);
179 }
180 else
181 {
182 //建立叶子节点
183 TreeNode* treeEnd = new TreeNode();
184 treeEnd->m_sAttribute = statTree[attr_node]->attriName;
185 treeEnd->m_iDeciNum = deci_Num;
186 treeEnd->m_iUnDecinum = sum - deci_Num;
187 treeNote->m_vChildren.push_back(treeEnd);
188 //打印叶子
189 if (deci_Num == 0)
190 {
191 cout << "叶子——" << treeLine << ">no" << " " << sum << endl;
192 }
193 else
194 {
195 cout << "叶子——" << treeLine << ">yes" << " " << deci_Num <<
        endl;
196 }
197 }
198 }
199 //还原属性列可用性
200 readClumNum[attr_node] = 0;
201 }
202
203 }
204 /*
205 * @function compuDecisiNote 计算C4.5
206 * @param: statTree 为状态树，此树动态更新，但是由于是DFS对数据更
```

```
    新，所以不必每次新建状态树
207 * @param: deciNum Yes的数据量
208 * @param: lineNum 计算set的行数
209 * @param: readClumNum 用于计算的set
210 * @return int 信息量最大的属性号
211 */
212 int DecisionTree::compuDecisiNote(vector<attributes*>& statTree,
        int deciNum, int lineNum, vector<int>& readClumNum)
213 {
214 double max_temp = 0;
215 int max_attribute = 0;
216 //总的yes行的信息量
217 double infoD = info_D(deciNum, lineNum);
218 for (int i = 0; i < attriNum - 1; i++)
219 {
220 if (readClumNum[i] == 0)
221 {
222 double splitInfo = 0.0;
223 //info
224 double info_temp = Info_attr(statTree[i]->attriItem, splitInfo,
      lineNum);
225 statTree[i]->statResult.push_back(info_temp);
226 //gain
227 double gain_temp = infoD - info_temp;
228 statTree[i]->statResult.push_back(gain_temp);
229 //split_info
230 statTree[i]->statResult.push_back(splitInfo);
231 //gain_info
232 double temp = gain_temp / splitInfo;
233 statTree[i]->statResult.push_back(temp);
234 //得到最大值*/
235 if (temp > max_temp)
236 {
237 max_temp = temp;
238 max_attribute = i;
239 }
240 }
```

```
241 }
242 return max_attribute;
243 }
244 /*
245 * @function Info_attr info_D 总信息量
246 * @param: deciNum 有效信息数
247 * @param: sum 总信息量
248 * @return double 总信息量比例
249 */
250 double DecisionTree::info_D(int deciNum, int sum)
251 {
252 double pi = (double)deciNum / (double)sum;
253 double result = 0.0;
254 if (pi == 1.0 || pi == 0.0)
255 {
256 return result;
257 }
258 result = pi * (log(pi) / log((double)2)) + (1 - pi)*(log(1 - pi)
      / log((double)2));
259 return -result;
260 }
261 /*
262 * @function Info_attr 总信息量
263 * @param: deciNum 有效信息数
264 * @param: sum 总信息量
265 * @return double
266 */
267 double DecisionTree::Info_attr(map<string, attrItem*>& attriItem
     , double& splitInfo, int lineNum)
268 {
269 double result = 0.0;
270 for (map<string, attrItem*>::iterator item = attriItem.begin();
271 item != attriItem.end();
272 ++item
273 )
274 {
275 double pi = (double)(item->second->itemNum[0]) / (double)lineNum
```

```
    ;
276 splitInfo += pi * (log(pi) / log((double)2));
277 double sub_attr = info_D(item->second->itemNum[1], item->second
    ->itemNum[0]);
278 result += pi * sub_attr;
279 }
280 splitInfo = -splitInfo;
281 return result;
282 }
283 /*
284 * @function resetStatTree 清理状态树
285 * @param: statTree 状态树
286 * @param: readClumNum 需要清理的属性set
287 * @return void
288 */
289 void DecisionTree::resetStatTree(vector<attributes*>& statTree,
    vector<int>& readClumNum)
290 {
291 for (int i = 0; i < readClumNum.size() - 1; i++)
292 {
293 if (readClumNum[i] == 0)
294 {
295 map<string, attrItem*>::iterator it_end = statTree[i]->attriItem
    .end();
296 for (map<string, attrItem*>::iterator it = statTree[i]->
    attriItem.begin();
297 it != it_end; it++)
298 {
299 delete it->second;
300 }
301 statTree[i]->attriItem.clear();
302 statTree[i]->statResult.clear();
303 }
304 }
305 }
306
307
```

```
308 \\Tree.cpp
309 #include "stdafx.h"
310 #include "Tree.h"
311
312 TreeNode* CreateTreeNode(string value)
313 {
314 TreeNode* pNode = new TreeNode();
315 pNode->m_sAttribute = value;
316 return pNode;
317 }
318
319 bool FindNode(TreeNode* pRoot, std::string& item)
320 {
321
322 if (pRoot->m_sAttribute == item)
323 return true;
324
325 bool found = false;
326
327 vector<TreeNode*>::iterator i = pRoot->m_vChildren.begin();
328 while (!found && i < pRoot->m_vChildren.end())
329 {
330 found = FindNode(*i, item);
331 ++i;
332 }
333
334 return found;
335 }
336
337 void ConnectTreeNodes(TreeNode* pParent, TreeNode* pChild)
338 {
339 if (pParent != NULL)
340 {
341 pParent->m_vChildren.push_back(pChild);
342 }
343 }
344
```

```
345 void PrintTreeNode(TreeNode* pNode)
346 {
347 if (pNode != NULL)
348 {
349 printf("value of this node is: %d.\n", pNode->m_sAttribute);
350 printf("its children is as the following:\n");
351 std::vector<TreeNode*>::iterator i = pNode->m_vChildren.begin();
352 while (i < pNode->m_vChildren.end())
353 {
354 if (*i != NULL)
355 printf("%s\t", (*i)->m_sAttribute);
356 ++i;
357 }
358 printf("\n");
359 }
360 else
361 {
362 printf("this node is null.\n");
363 }
364
365 printf("\n");
366 }
367
368 void PrintTree(TreeNode* pRoot)
369 {
370 PrintTreeNode(pRoot);
371
372 if (pRoot != NULL)
373 {
374 std::vector<TreeNode*>::iterator i = pRoot->m_vChildren.begin();
375 while (i < pRoot->m_vChildren.end())
376 {
377 PrintTree(*i);
378 ++i;
379 }
380 }
381 }
```

```
382
383 void DestroyTree(TreeNode* pRoot)
384 {
385 if (pRoot != NULL)
386 {
387 std::vector<TreeNode*>::iterator i = pRoot->m_vChildren.begin();
388 while (i < pRoot->m_vChildren.end())
389 {
390 DestroyTree(*i);
391 ++i;
392 }
393 delete pRoot;
394 }
395 }
```