

《人工智能实验》

实验报告

(实验二)

学 院 名 称 : 数据科学与计算机学院

专业 (班级) : 16 计科 2 班

学 生 姓 名 : 李新锐

学 号 : 15323032

时 间 : 2018 年 10 月 9 日

一、实验题目

决策树

二、实验内容

(一)、算法原理

决策树是一种基于树结构的非线性分类模型。

结构上看，决策树的根节点和中间节点表示对该节点有关的数据集合 D 依据特征空间 A 中的一个特征 a 进行分类。节点通过边和子节点相连，每一条边关联 a 的一种取值。叶子节点的值表示最终的分类结果。

从特征空间的角度上看，决策树的建树过程相当于对特征空间不断使用直线（或高维的超平面）进行划分。由于决策树可以对多个特征逐层划分，相当于可以画出很多条直线，从而可以解决非线性的分类问题。

决定决策树分类效果的关键在于选择哪个标准进行分类，根据这一点不同衍生出了ID3，C4.5，CART三种决策树；以及剪枝方法，常见有前剪枝和后剪枝。

(二)、伪代码

1. 建树过程：该过程是一个递归过程，伪代码如下：

```
1  # 依据训练样本集D和特征集A训练决策树
2  # judge_func是一个函数参数，它的不同决定了决策树是ID3/C4.5还是CART
3  train(D, A, judge_func):
4      # 训练集D的X, Y分开
5      (X, Y) = D
6      # 记mode为当前节点数据集的结果的众数
7      mode = sum(Y) > len(Y) / 2
8      '''
9      结束条件有4种，分别是：
10     1. 训练集为空
11     2. 特征集为空
12     3. 所有训练数据特征均一致
13     4. 所有训练数据分类一致
14     '''
15     if empty(D) or empty(A) or all_equal(X) or all_equal(Y):
16         # 设置当前节点为叶节点，结果为mode
17         node.isLeaf = True
18         node.Y = mode
19         return node
```

```

20     # 对于每一种特征计算信息增益或基尼系数的减少值
21     info_gain = [judge_func(D, a) for a in A]
22     # a* 是信息增益最大的一组特征
23     a* = indexof(A, max(info_gain))
24     # 在特征空间中移除a*
25     erase(A, a*)
26     # 对于a*的每个取值
27     for i in featureValues[a*]:
28         # 取对应属性相同的子训练集
29         S = subset(D, a*)
30         # 递归训练并添加到子节点集中
31         node.child.append(train(S, A))

```

2. 计算信息增益或基尼系数的减少量的函数

首先定义计算概率为 p 的二项分布的熵函数和基尼系数函数

```

1  H(p): - p * log(p) - (1-p) * log(1-p)
2  gini(p): 2 * p * (1-p)

```

ID3, C4.5, CART的有关信息增益或基尼系数的减少值的计算过程基本相同, 可以抽象出一个 judge_func_base函数

```

1  judge_func_base(D, a, Fun):
2      (X, Y) = D
3      # 由于Y的取值仅为0或1, 故sum(Y)即是Y中1的个数
4      p = sum(Y) / len(Y)
5      # 依据特征划分前的熵
6      H_D = H(p)
7      # 依据条件a划分条件熵
8      H_D_When_a = 0
9      # 依照公式, 对a的每种取值算出p * Fun(p)求和
10     for i in a:
11         (SX, SY) = subset(D, i)
12         p = len(SY) / sum(SY)
13         H_D_When_a += p * Fun(p)
14     return H_D - H_D_When_a

```

基于它们可以分别定义ID3, C4.5, CART为

```

1  ID3(D, a): judge_func_base(D, a, H)
2  C45(D, a): judge_func_base(D, a, H) / split_info(D, a)
3  CART(D, a): judge_func_base(D, a, gini)

```

其中split_info(D, a)定义为

```

1  # 依照公式求和
2  split_info(D, a):
3      for i in a:
4          (SX, SY) = subset(D, i)
5          p = len(SX) / len(D)
6          SplitInfo += - p * log(p)

```

3. 预测函数

在建好树后即可对新的数据进行预测，预测代码很简单：

```
1 predict(x):
2     # 从根节点开始遍历
3     head = root
4     # 如果没有遍历到叶节点就一直遍历
5     while head is not leaf:
6         # 取当x的用于划分当前节点的特征的值
7         c = X[head.a]
8         # 依据该值选择子节点
9         head = head.child[c]
10    # 返回叶子节点的值
11    return head.y
```

(三)、关键代码截图（带注释）

1. 建树函数：

实际代码是用C++编写，和上述伪代码逻辑是基本一致的，区别在于对D为空的判断放在递归步判断，这样可以节省一次不必要的递归。决策树相关函数实现在DecisionTree类中。

```
1  /*
2  * 对应伪代码中的train函数
3  * Node为节点类型
4  * judge_func是一个函数参数，它的不同决定了决策树是ID3/C45还是CART
5  */
6  void DecisionTree::train_worker(DecisionTree::Node *node, const JudgeFunc_t &judgeFunc)
7  {
8      //获取该节点全部数据集的Y值
9      auto D_Y = node->D.row(-1);
10     //计算众数
11     auto mode = (double)D_Y.sum() >= (node->D.cols() / 2.0);
12     //若A为空集
13     if(node->A.empty())
14     {
15         node->isLeaf = true;
16         node->Y = mode;
17         return;
18     }
19     //若D中全部样本属于同一类型
20     if(all_equal(D_Y))
21     {
22         node->isLeaf = true;
23         node->Y = D_Y[0];
24         return;
25     }
26     //若所有样本各个属性值均相同
```

```

27     bool flag = true;
28     for(Index i = 0; i < node->D.rows() - 1; ++i)
29     {
30         if(!all_equal(node->D.row(i)))
31         {
32             flag = false;
33             break;
34         }
35     }
36     if(flag)
37     {
38         node->isLeaf = true;
39         node->Y = mode;
40         return;
41     }
42     //对于每一种特征计算信息增益（率）或基尼系数的减少值
43     Vec<double> info_gain;
44     for(auto i: node->A)
45         info_gain.push_back(judgeFunc(node->D, i, featureValues[i]));
46     //a_star是信息增益最大的一组特征
47     auto a_star = node->A[max_element(info_gain) - std::begin(info_gain)];
48     node->C = a_star;
49     //依据该组特征值划分数据集
50     Vec<Vec<Index>> S(featureValues[a_star]);
51     for(int i = 0; i < node->D.cols(); ++i)
52     {
53         S[node->D(a_star, i)].push_back(i);
54     }
55     //在特征空间中移除a_star
56     node->A.erase(ranges::v3::find(node->A, a_star));
57     //对于a_star的每个取值
58     for(int i = 0; i < featureValues[a_star]; ++i)
59     {
60         // 取对应属性值相同的子训练集
61         auto n = new Node(matrix_view<int>(node->D, S[i]), node->A);
62         // 若子训练集为空，则将这个子节点标为叶节点，无需递归下去
63         if(S[i].empty())
64         {
65             n->isLeaf = true;
66             n->Y = mode;
67             node->child.push_back(n);
68         }
69         else
70         {
71             //递归训练并添加到子节点集中
72             train_worker(n, judgeFunc);
73             node->child.push_back(n);
74         }
75     }

```

由于计算信息增益（率）或基尼系数的减少量的函数是简单的公式翻译的产物，和伪代码更没什么大区别，这里就不展示了

四、创新点&优化

一、由于决策树构建过程中经常要用取子数据集的操作，因此我实现了一个matrix_view（矩阵视图）类。它接收一个矩阵（代码中使用Eigen3矩阵）或一个父视图，和一个包含子数据集数据编号set，生成表示子数据集的视图。它仅储存编号而不需要拷贝矩阵的内容，从而提高了训练速度。同时在输入向量维度为 n 时，能够降低近 $\frac{n-1}{n}$ 的内存使用量。

二、实现了PEP（悲观错误剪枝）

悲观错误剪枝是一种后剪枝方法，它仅依赖训练集。它首先计算训练集在某个中间节点上的错误率 ec

$$ec = \frac{E + punish * L}{N}$$

其中 E 是分类错误的个数， $punish$ 是乘法系数， L 是中间节点的子孙中叶子节点的个数， N 的输入数据总个数。

假设样本以伯努利分布，则根据 ec 计算出误判次数的标准差

$$SD = \sqrt{N * ec * (1 - ec)}$$

而若将中间节点改为叶子节点，新的错误率 ec_{new} 为

$$ec_{new} = E_{new} + punish$$

若 $ec + SD > ec_{new}$ 则将该节点剪枝。

实现代码如下：

```

1 //PEP剪枝函数，返回值是当前节点下的叶子节点个数
2 int DecisionTree::prune_worker(Node* node)
3 {
4     //若当前节点是叶子节点，直接返回1
5     if(node->isLeaf)
6         return 1;
7     else
8     {
9         //统计叶子节点个数
10        int num_leaf = 0;
11        for(auto& i: node->child)
12            num_leaf += prune_worker(i);
13        //统计当前训练集上的错误个数error
14        auto ret = predict(node->D);
15        auto num_X = node->D.cols();
16        double error = 0;
17        for(Eigen::Index i = 0; i < num_X; ++i)
18            if(node->D(-1, i) != ret[i])

```

```

19         error++;
20         //计算错误率ec
21         constexpr double punish = 0.5;
22         auto ec = (error + punish * num_leaf) / num_X;
23         //假设样本为伯努利分布，计算标准差SD
24         auto SD = std::sqrt(num_X * ec * (1-ec));
25         //计算剪枝后的错误个数new_error
26         auto D_Y = node->D.row(-1);
27         auto mode = (double)D_Y.sum() >= (node->D.cols() / 2.0);
28         double new_error = 0;
29         for(Eigen::Index i = 0; i < num_X; ++i)
30             if(node->D(-1, i) != mode)
31                 new_error++;
32         //若剪枝后错误率更低，则剪枝
33         if(error + SD > new_error + punish)
34         {
35             //将当前节点设为叶子节点，结果设为众数
36             node->isLeaf = true;
37             node->C = -1;
38             node->Y = mode;
39             //删除子节点
40             for(auto &i: node->child)
41                 delete(i);
42             node->child.clear();
43             //返回叶子节点数为1
44             return 1;
45         }
46         //否则不剪枝
47         else
48             return num_leaf;
49     }
50 }

```

五、实验结果展示

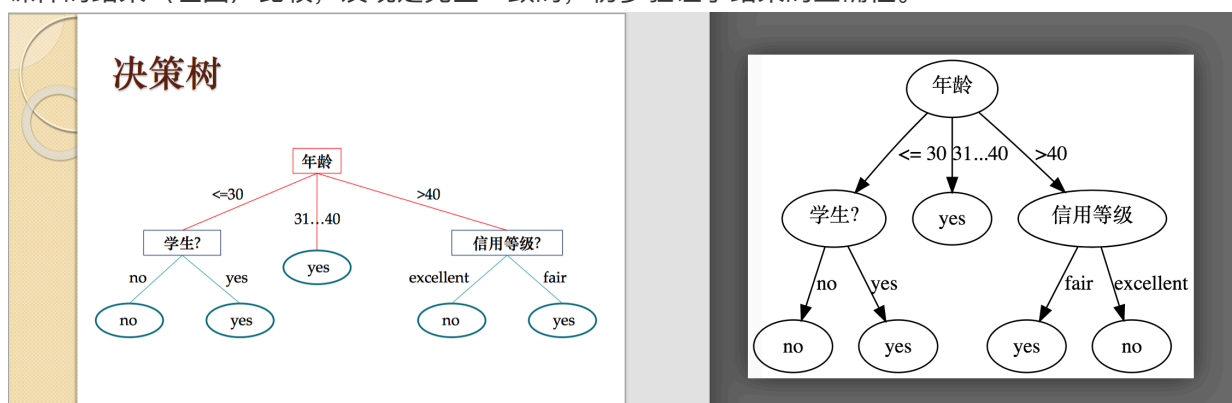
1.使用理论课课件的小数据集测试

理论课课件中提供了一个关于是否购买电脑的小数据集的ID3决策树样例（如下图）：

决策树

年龄	收入	学生?	信用等级?	是否买电脑
<=30	high	no	fair	no
<=30	high	no	excellent	no
31...40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31...40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31...40	medium	no	excellent	yes
31...40	high	yes	fair	yes
>40	medium	no	excellent	no

我使用graphviz库实现了生成的决策树的可视化，将我使用ID3算法得到的结果（下图右边）与理论课件的结果（左图）比较，发现是完全一致的，初步验证了结果的正确性。



2. 训练集测试

以完整的训练集作为输入，我分别测试了ID3，C4.5，CART三种算法，生成的完整树很宽，因此保存为了PDF文件。这里仅展示部分效果

ID3:

注意到本实验给出的样本的特殊性：给出的特征足够丰富，导致从输入样本 X 到分类结果 Y 是一个单射，训练集在生成的决策树上的准确率为1，因此PEP剪枝算法不会剪掉任何一个节点，因此结果展示和下面评测分析中不区分是否采用该方法剪枝（结果是完全一样的）。但在实际中，给出的特征常常是不足的，PEP算法对于这种情况有着防止过拟合的价值。

六、评测指标展示即分析

我使用了K交叉验证方法，即从样本分割为K份，依次选取其中一份作为验证集，其余数据作为训练集。我遍历了K从2到8的情况。验证用代码如下（C++，range函数是通过封装了range-v3库提供的view::ints函数实现）

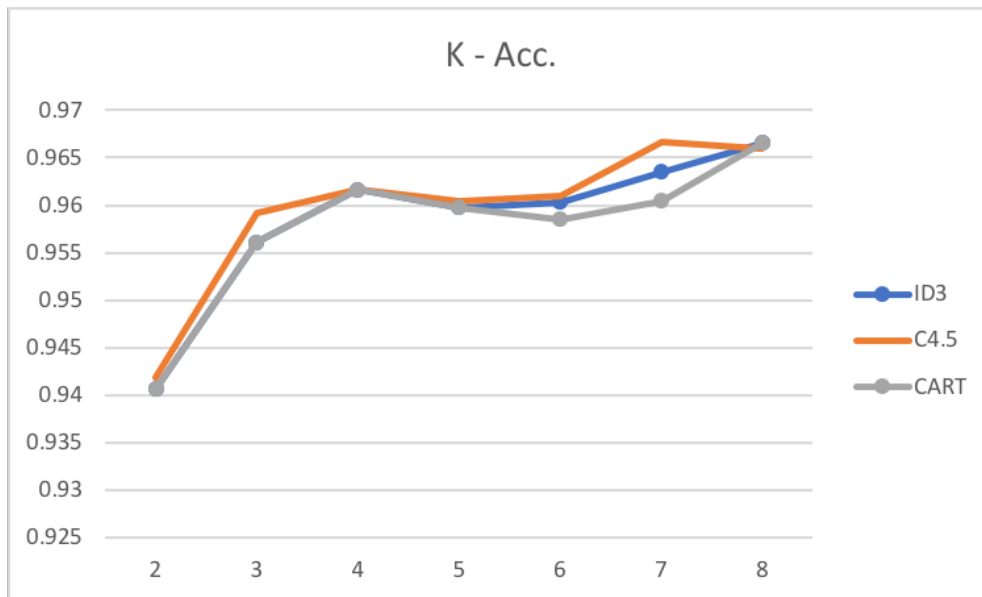
```
1 //读取训练集数据
2 auto f = readFile("data/Car_train.csv");
3 //向量化
4 auto data = vectorizeData(f, mp);
5 //K遍历2到8的值
6 for(double K: range(2, 9))
7 {
8     //N是总输入数据数
9     int N = data.cols();
10    int pieceSize = N / K;
11    auto all = range(0, N);
12    for(auto i : range(0, K))
13    {
14        //分割验证集 1/K 的数据作为验证集
15        auto vaildSetRange = range(i * pieceSize, (i + 1) * pieceSize);
16        matrix_view<int> vaildSet(data, vaildSetRange);
17        //其余数据作为训练集
18        auto trainSetRange = view::set_difference(all, vaildSetRange);
19        matrix_view<int> trainSet(data, trainSetRange);
20        //遍历三种决策树方法
21        for(auto& [name, Func] : JudgeFuncs)
22        {
23            //各个特征的取值可能数
24            auto featureValRanges = {4,4,4,3,3,3};
25            DecisionTree t(trainSet, featureValRanges);
26            //计时并建树
27            auto start = now();
28            t.train(Func);
29            auto diff = now() - start;
30            //验证并打印结果
31            auto acc = t.vaild(vaildSet);
32            print_ret(K, name, diff, acc);
33        }
34    }
35 }
```

验证结果如下：

如图，横坐标是不同的K值，纵坐标是确定一个K值后，进行K此交叉验证的平均分类准确率，

横向比较K值和准确度的关系，我们可以观察到，除K取2时准确度相对较低之外，K取3到8时准确度均在 $96\% \pm 0.6\%$ 的区间内。我认为这是由于K=2时没有足够的信息来支持决策树的建立。观察到K=4时是一个极值点，再增加时反而降低，这可能是过拟合引起的。K>5时准确度再次上升，然而此时验证集已经越来越小，所得的结果不能很好反映决策树的预测能力。

纵向比较来看，C4.5的准确率一直是三者中最高的。



思考题

1. 决策树有哪些避免过拟合的方法？

答：一、样本角度：对样本进行筛选，排除错误值和离群值。

二、决策树角度：采用剪枝方法，包括预剪枝和后剪枝。前者是在建树时限制决策树的高度和叶子节点的数目。后者是通过用叶子节点代替一些置信度不够的中间节点。

2. C4.5相比于ID3的优点是什么，C4.5又可能有什么缺点？

答：ID3倾向于选择分支较多的特征，考虑一种极端情况，样本集合D中每个元素的特征a均不同，那么ID3就会直接选择这一特征并完成建树。这样得到的树是严重过拟合的。C4.5通过引入SplitInfo和信息增益率，每次分割涉及的数据越多SplitInfo越大，SplitInfo作为分母，信息增益率就越小。从而避免了过拟合。缺点是运算量稍大。

3. 如何用决策树来进行特征选择（判断特征的重要性）？

决策树总是试图选择能将同类节点整合在一起，选择能够最大程度降低分类不确定性的特征，因此自然采用熵族函数（满足上凸并在两端为0）对特征的重要性进行评估。具体有基于信息增益（互信息）的ID3算法，基于信息增益率（互信息除以分割熵）的C4.5算法和基于gini系数的CART算法。