决策树算法的实现

17341146 王程钥

1 算法原理

决策树是一类常见的机器学习方法，它将分类问题看作分步决策的过程，通过一步一步的判断和决策获得最终的分类结果。顾名思义，决策树是基于树的结构工作的，它通过构建一棵决策树来完成决策。

一棵决策树包含一个根节点，若干内部节点和若干叶子节点。叶子结点对应决策结果，其他每个节点对应一个属性的测试。根节点包含样本全集，每个中间节点包含样本子集，并通过属性测试的结果将样本划分到若干子节点中。从根节点到叶子节点的一条路景对应了一次决策的过程。

决策树是一个递归的过程。在训练的时候，若递归到的节点的所有样本属于同一类别，或所有样本在所有属性上取值为空，或属性集为空，则会结束递归并打标签返回。否则，会在属性集中选择一个最优秀的属性，基于该属性划分叶子结点，并继续往下递归。

属性划分的常见估价函数有信息增益 `gain(D,A)`，信息增益率 `gain_ratio(D,A)`，`Gini` 系数 `gini(D,A)`。

$$\mathrm{gain}(\mathrm{D},\mathrm{A}) = \mathrm{H}(\mathrm{D}) - \mathrm{H}(\mathrm{D}|\mathrm{A}) \qquad (1)$$

$$\mathrm{gain}_{\mathrm{ratio(D,A)}} = \frac{\mathrm{H}(\mathrm{D}) - \mathrm{H}(\mathrm{D}|\mathrm{A})}{splitinfo(D,A)} \qquad (2)$$

$$splitinfo(D,A) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} * \log(\frac{|D_j|}{|D|}) \qquad (3)$$

$$\mathrm{gini}(\mathrm{D},\mathrm{A}) = \sum_{j=1}^{v} p(A_j) * gini(D_j|A = A_j) \qquad (4)$$

$$\mathrm{gini}(D_j|A = A_j) = 1 - \sum_{i=1}^{n} p_i^2 \qquad (5)$$

在测试的时候，样本根数决策树上每个节点的属性和自己的特征，从根节点往下递归，直到递归到叶子节点或找不到对应的属性值，并返回递归结束节点的标签作为预测结果。

2 算法流程

Algorithm1 decision_tree
Input: 训练集 Train_x, Train_y, 属性集 A **Output:** 决策树 Tree 生成节点 node 根据 Train_y 中样本数最多的类获取 node 节点的 label 给节点 node 标记上 label if Train_y 全部属于同一类别 or A 为空 or Train_x 全部相同 then 　　将 node 节点标记为叶子节点; return

```
end if
从训练集(Train_x, Train_y)中获取最优划分属性 a
在 A 中删去 a
for a 的在(Train_x, Train_y)中的每一个可能取值 ai do:
    为该分支创立分支节点 node’
    Train_xi, Train_yi = 在属性 a 上取值为 ai 的样本集
    Tree.append( decision_tree(node’, Train_xi, Train_yi, A) )
end for
return Tree
```

算法 1 为递归构建决策树的伪代码，输入训练集和属性集，返回决策树

```
Algorithm2 get attribute
Input: 训练集 Train_x, Train_y, 属性集 A
Output: 最优划分属性 best_a
For a in A do:
    将 Train_x, Train_y 根据 ai 划分成若干子集 Train_xi, Train_yi
    entropy=-Σp(ai)*(p0*log(p0)+p1*log(p1))
    if entropy>max_entropy then
        max_entropy=entropy; best_a=a
return best_a
```

算法 2 为获取最优划分属性的算法，输入训练集和属性集，返回最优属性。以 ID3 算法为例，C4.5 算法和 CART 算法类似。

3 代码解析

```
def split_data(data_x,data_y,K):
    size = len(data_x)
    block = (size+K-1)//K
    res_x = []; res_y = []
    for i in range(K):
        pos = list(range(i*block,min((i+1)*block,size)))
        res_x.append(data_x[pos])
        res_y.append(data_y[pos])
    res_x = np.array(res_x)
    res_y = np.array(res_y)
    return res_x,res_y
# K 折交叉验证数据集分割
```

K 折交叉验证分割训练集与验证集

```
def calc(data,method='entropy'):
    len_data = len(data)
    label_list = set()
    if method=='gini': result = 1
    else : result=0
```

```
    for index in data:
        label_list.add(index)
    for index in label_list:
        size = len([y for y in data if y==index])
        P = size/len_data
        if method=='entropy': result -= P*log2(P)
        else : result -= P*P
    return result
# 信息熵 or Gini 指数

def calc_splitinfo(data):
    entropy = 0; cnt_D = 0
    for index in data:
        cnt_D += len(index)
    for index in data:
        P = len(index)/cnt_D
        entropy -= P*log2(P)
    if entropy==0: return 1e-6
    return entropy
# 属性混乱度
```

信息熵，Gini 指数和属性混乱度的计算的计算

```
def training(now, train_x, train_y, valid_x, valid_y,
             attribute, method='ID3', val='yes', C=2.5, depth=10000 ):
    global cnt_node
    H=calc(train_y,method='entropy')
    label=get_label(train_y)
    wrong_train = len([x for x in train_y if x!=label])
    wrong_valid = len([x for x in valid_y if x!=label])

    if len(attribute)==0 or H==0 or equal(train_x) or depth<=0:
        return [(now,[],0,0,label)], wrong_train, wrong_valid, 1
    # 终止条件

    len_data = len(train_x)
    feature=''; max_h=-10000
    # 初始化

    for item in attribute:
        now_h=0
        new_x,new_y,_ = split_set(train_x, train_y, item)
        for now_x,now_y in zip(new_x,new_y):
            P=len(now_x)/len_data
            if method=='CART': now_h += P*calc(now_y,method='gini')
            else: now_h += P*calc(now_y,method='entropy')
```

```
            # 计算条件信息熵 or Gini 指数
        if method=='ID3': now_h=H-now_h # 信息增益
        if method=='C4.5': now_h = (H-now_h)/calc_splitinfo(new_x)
        # 信息增益率
        if method=='CART': now_h = -now_h # gini 最小 -> -gini 最大
        if now_h>max_h: max_h=now_h; feature=item
    # 寻找最优特征

    tree=[]; child_now=[]; train_split=0; valid_split=0; leaves_size=0
    attribute.remove(feature)
    tri_x, tri_y, feature_set = split_set(train_x, train_y, feature)
    # 分割数据集

    for i in range(len(tri_x)):
        cnt_node+=1; val_x=[]; val_y=[]
        child_now.append(cnt_node)
        if val=='yes':
            for x,y in zip(valid_x, valid_y):
                if x[feature]==feature_set[i]:
                    val_x.append(x); val_y.append(y)
        tree_son, train_son, valid_son, leaves =
            training(cnt_node, tri_x[i], tri_y[i], val_x, val_y,
            attribute.copy(),method=method,val=val, C=C, depth=depth-1)
        train_split += train_son
        valid_split += valid_son
        leaves_size += leaves
        tree.extend(tree_son)
    # 递归

    node = (now,child_now,feature,feature_set,label)

    if val=='yes':
        if valid_split > wrong_valid:
            node = (now,[],0,0,label); leaves_size=1
            train_split = wrong_train
        else: wrong_valid = valid_split
    # 后剪枝
    if train_split+C*leaves_size > wrong_train + C:
        node = (now,[],0,0,label); leaves_size=1
    else: wrong_train = train_split
    # 泛化错误率

    tree.append(node)
    return tree, wrong_train, wrong_valid, leaves_size
```

```
# 训练，深搜建树
```
训练生成决策树
```
def testing(now,train_x,tree):
    _, child, index, feature_set, label = tree[now-1]
    if child==[]: return label
    for ch, item in zip(child, feature_set):
        if train_x[index]==item:
            return testing(ch, train_x, tree)
    return label
# 测试
```
在决策树上测试

4 优化点与部分说明

4.1 训练集和验证集的划分

本次实验我采取的 5 折交叉验证的方法，即将训练集分成 5 份，每次枚举一份作为验证集，其余四份作为训练集。使用训练集构建决策树，使用验证集对训练集构成的决策树进行测试。将 5 折交叉验证的准确率取平均作为训练结果的评估。

4.2 基于泛化错误率的剪枝

我们尝试融入模型复杂度，引入泛化错误率，对决策树进行正则化。我们定义一棵子树的错误率为$\mathrm{e}=\frac{\sum r(i)}{\sum m(i)}$，其中 r(i)表示子树 i 中错误分类的样本数，m(i)表示子树 i 的总样本数。

基于错误率，我们又定义了泛化错误率为$e_c=\frac{\sum r(i)+l(i)*C}{\sum m(i)}$，其中 l(i)表示子树 i 的叶子数，C 表示罚项。

我们使用了后剪枝的方法，即在递归建树完成后向上回溯的过程中，如果对于当前节点 node 的子树，将 node 作为叶子的泛化错误率小于原子树的泛化错误率，则将 node 置为叶子结点，并删除它下面的子树。

4.3 基于最大深度的剪枝

这是一种简单的预剪枝操作。通过限制树的深度，降低决策树的模型复杂度，以期在验证集上取得更好的效果。

4.4 基于验证集的剪枝

我们尝试在训练集中再分出 1/10 的数据作为验证集，将其代入决策树的递归，边建树边根据验证集剪枝。我们采取了后剪枝的方法，即在回溯的过程中，如果将 node 节的子树都剪枝掉并将其置为叶子节点可以使验证集的准确率提升，则将 node 节点的子树剪枝。

同时，基于验证集的剪枝可以和 4.2 节和 4.3 节中基于模型复杂度的剪枝相结合。实验证明这样可以获得更好的效果。

5 实验结果

5.1 标准决策树算法的结果

我分别使用了 ID3，C4.5 和 CART 算法，对训练集按照原顺序分为 5 折进行交叉验证。如

表 1 所示，三种算法的分类准确率是完全一样的。我认为这是由于训练集样本总体类别不多，每个特征都只有 3 类或 4 类，属性的混乱度不大，无论采用哪种度量方式选出的最优属性都相同导致的。

模型	第 1 组	第 2 组	第 3 组	第 4 组	第 5 组	平均
ID3	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%
C4.5	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%
CART	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%

(表 1)

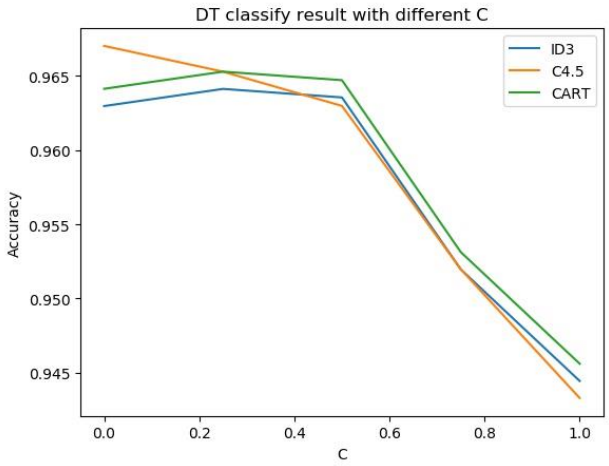
出于对算法鲁棒性的测试，我尝试将训练集打乱重新分组进行 5 折交叉验证，结果如表 2 所示，三种算法的准确率都达到 97%以上且不再出现相同。可以看到 C4.5 的准确率略高于 ID3，ID3 又略高于 CART。

模型	第 1 组	第 2 组	第 3 组	第 4 组	第 5 组	平均
ID3	97.69%	97.69%	96.24%	97.40%	96.51%	97.11%
C4.5	97.69%	97.69%	96.24%	97.40%	97.09%	97.57%
CART	97.69%	97.69%	96.24%	97.40%	95.06%	96.82%

(表 2)

5.2 基于泛化错误率的剪枝

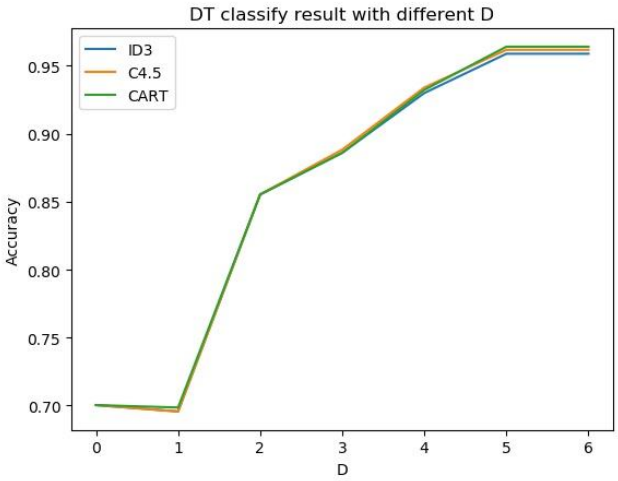
我分别使用三种算法，在不加其他优化的情况下，在[0,1]的范围内枚举 C 值，使用随机打乱的数据进行 K 折交叉验证。如图，可以看到在使用 C4.5 和 ID3 算法时，当 C=0.25 时验证集上的准确率最高。由此可见，在选择一个较小的 C 时，使用基于泛化错误率的剪枝可以给算法带来一定的提升。



(图 1)

5.3 基于最大深度的剪枝

我使用三种算法，在不加其他优化的情况下，在[0,5]的范围内枚举最大深度(根节点在第 0 层)，使用随机打乱的数据进行 K 折交叉验证。如图 3 可以看到，基于最大深度的剪枝其实对准确率的提升并不明显。这主要是由于本次的训练集样本种类不多。除了提高准确率外，限制深度还可以起到减少训练时间，提高效率的目的。



(图 2)

5.4 基于验证集的剪枝

我使用 ID3，C4.5 和 CART 三种算法，在不加其他优化的情况下，使用不进行打乱的数据进行 K 折交叉验证。使用了基于验证集的剪枝后，使用 ID3 算法和 C4.5 算法的准确率有所下降，而使用 CART 算法的准确率有所提升。准确率下降的主要原因是，我们取了训练集中部的部分数据作为验证集，也就减少了训练样本的数量。由此可见，基于验证集的剪枝还是有一定作用的。

method	Cut	Fold1	Fold2	Fold3	Fold4	Fold5	Avg
ID3	no	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%
ID3	yes	96.82%	98.55%	97.40%	95.95%	95.93%	96.93%
C4.5	no	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%
C4.5	yes	96.82%	98.27%	97.11%	95.95%	97.09%	97.05%
CART	no	96.82%	98.55%	97.40%	96.53%	96.22%	97.11%
CART	yes	96.82%	98.55%	97.40%	95.95%	97.09%	97.16%

(表 3)

5.5 最优结果

通过枚举三种不同的算法，并将三种剪枝方式进行结合，我们最终发现，在不随机打乱测试集的情况下，使用 CART 算法，C=0，采用基于验证集的剪枝，不限制最大深度时 5 折交叉验证的平均准确率最高，达到了 97.16%。

	Fold1	Fold2	Fold3	Fold4	Fold5	Avg
训练集	100%	100%	100%	100%	100%	100%
验证集	98.56%	98.56%	98.56%	99.28%	97.84%	98.56%
测试集	96.82%	98.55%	97.40%	95.95%	97.09%	97.16%

(表 4)

6 思考题

6.1 决策树有哪些避免过拟合的方法

1）对决策树进行剪枝； 2）引入罚项进行正则化 3）限制决策树的深度

6.2 C4.5 相比于 ID3 的优点是什么

和 ID3 相比，C4.5 多考虑的属性的混乱度，它倾向于选择信息增益高且混乱度低的属性。这可以防止高混乱度的属性带来的过拟合。

6.3 如何用决策树来进行特征选择

枚举每个特征，按照该特征分割数据集，通过特定指标判断分割后结果的好坏。
一种简单的方法是使用加权准确率，即对于每个分支采用多数投票法确定标签计算准确率，并根据样本数量加权计算分割准确率。
ID3 算法采用的方法是计算信息增益率，即对标签的概率计算信息熵，找到信息增益最大，即对标签混乱度降低最大的特征。
C4.5 算法在 ID3 算法的基础上，引入了信息增益率，使得属性的划分不会过度倾向种类更多的属性。
CART 算法使用的估价函数是 GINI 指数，通过 GINI 指数评估分割后标签的混乱度，选择人混乱度最小的属性作为这一层的特征。

7 结论

总体而言，在向本次实验这样属性较少的二分类任务上决策树的表现还是非常优秀的，总体来说可以达到 97%左右的准确率。因为本次实验的数据集较为简单，所以剪枝对准确率的提升并不明显，有时候还会带来反作用。但在更复杂的数据集上，决策树很容易出现过拟合，这是剪枝操作就会派上用场了。