

人工智能lab1实验报告

学号: 17338233 专业: 计科 姓名: 郑戈涵

决策树

算法原理

决策树是一种有监督学习的方法, 其样本有自己的属性和分类的结果。学习过程中, 将会根据样本的属性和分类结果不断地建立分支和节点, 形成一棵树。一般通过递归的方式生成。递归的终止条件有三种情况:

- 当前节点的样本的分类结果一样
- 样本虽然分类结果不一致, 但是没有属性可以用来区分样本
- 样本集合为空

测试时, 决策树通过判断样本的属性不断进入分支, 到叶节点时可以得到分类的结果。

由于分支取决于属性, 而不同的属性顺序会导致树的长度不同, 因此决定分支属性的不同方法会影响算法的效率。这次实验共有三个算法:

- ID3
 - 基于信息增益选择决策点
 - 步骤
 1. 计算数据集D的经验熵

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

2. 计算特征A对数据集D的条件熵 $H(D|A)$

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

3. 计算信息增益

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

4. 选择信息增益最大的特征作为决策点

- C4.5
 - 基于信息增益率选择决策点
 - 步骤

1. 计算A对数据集D的信息增益

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

2. 计算数据集D关于特征A的值的熵 $SplitInfo(D, A)$

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

3. 计算信息增益率

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

4. 选择信息增益率最大的特征作为决策点

- CART

- o 基于基尼系数选择决策点
- o 步骤
 1. 计算特征A的条件下，数据集D的基尼系数

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

其中:

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

v表示A的取值个数，n表示类别个数

2. 选择基尼系数最小的特征作为决策点

伪代码或者流程图

主模块

```

1  proc decision_tree():
2  input:  dataSet,
3          k, //交叉验证参数
4          method //算法(ID3, C4.5, CART)
5  output tree, accuracy
6          train_set, validation_set=k_fold(dataSet, k)
7          tree=CreateTree(train_set)
8          accuracy<-accuracy+validate(tree, validation_set)
9          return accuracy/k
10 end proc

```

验证模块

```

1  proc validate():
2  input:  tree, validation_set
3  output: accuracy
4          corrCnt=0
5          for sample in validation_set:
6              if classify(tree, sample)==validation_set[sample].result
7                  corrCnt<-corrCnt+1
8              end if
9          end for
10         return corrCnt/validation_set.size
11 end proc

```

分类模块

```

1  proc classify():
2      input:  tree,target
3      output: result
4          if tree.child!=None:
5              return classify(tree.child[target[tree.feature]],target)
6          else:
7              return tree.result
8          end if
9      end proc

```

代码展示

本次编程使用python的面向对象方法，算法和决策树对象有较好的解耦。

条件熵

取出数据代入公式即可。

```

1  from math import log
2  from collections import Counter
3  log2=lambda x:log(x)/log(2)
4  def rel_entropy(dataSet,feat):
5      # 投影取出特征和标签列
6      currfeat=dataSet.loc[:, [feat, 'Label']]
7      # 统计特征的个数
8      featCount=Counter(currfeat.iloc[:,0])
9      part_ent=0.0
10     ent=0.0
11     for feature in featCount:
12         proportion=featCount[feature]/dataSet.shape[0]
13         LabelCnt=Counter(currfeat.loc[currfeat[feat]==feature]['Label'])
14         totalCnt=sum(LabelCnt.values())
15         for r in LabelCnt.keys():
16             p=float(LabelCnt[r])/totalCnt
17     # 条件熵公式
18     part_ent=part_ent-p*log2(p)
19     # 加权求和
20     ent+=part_ent*proportion
21     part_ent=0
22     return ent

```

经验熵

本函数同时可以用于计算C4.5的D关于某特征的熵，取出数据后代入公式即可

```

1  def entropy(dataSet, feature='Label'):
2      ent=0.0
3      exentCnt=Counter(dataSet.loc[:, feature])
4      for res in exentCnt:
5          p=float(exentCnt[res])/dataSet.shape[0]
6          ent-=p*log2(p)
7      return ent

```

最佳特征 (ID3,C4.5)

此函数需要区分两种方法，判断最佳特征的基准不同。

```
1 def getBestFeature(dataSet,method='C4.5'):
2     maxEntGain=0.0
3     BestFeat=None
4     entGain=0.0
5     exEnt=entropy(dataSet)
6     for feat in dataSet.columns[:-1]:
7         entGain=exEnt-rel_entropy(dataSet,feat)
8         if method=='ID3':
9             pass
10    # 需要除以经验熵
11    elif method=='C4.5':
12        entGain/=entropy(dataSet,feat)
13    if(maxEntGain<entGain):
14        maxEntGain=entGain
15        BestFeat=feat
16    return BestFeat
```

最佳特征 (CART)

特征的最佳分割值

CART使用特征的值进行分支，因此需选定一个特征的值计算基尼系数，通过比较不同值得到的基尼系数得到某个特征的最佳分割值

。

```
1 def getBestSplit(dataSet,feat):
2     # 基尼系数小于1
3     minGiniIndex=1.0
4     BestSplitVal=None
5     # 取得当前特征对应的值的集合
6     featLst=list(Counter(dataSet[feat]))
7     # 对每个值，计算一次基尼系数
8     for featVal in featLst:
9         currfeat=dataSet.loc[:,[feat,'Label']]
10        featCount=Counter(currfeat.iloc[:,0])
11        onesSubtractGini=0.0
12        giniIndex=0.0
13    # 将样本按照等不等于该值分成两组
14    target_group=currfeat.loc[currfeat[feat]==featVal]['Label']
15    another_group=currfeat.loc[currfeat[feat]!=featVal]['Label']
16    portion=target_group.size/currfeat.shape[0]
17    targetCnt=Counter(target_group)
18    anotherCnt=Counter(another_group)
19    p1=float(targetCnt[0])/target_group.size
20    p2=float(anotherCnt[0])/another_group.size
21    # 使用公式得到基尼系数
22    giniIndex+=2*(1-p1)*p1*portion+(2*(1-p2)*p2)*(1-portion)
23    if(giniIndex<minGiniIndex):
24        BestSplitVal=featVal
25        minGiniIndex=giniIndex
26    return BestSplitVal,minGiniIndex
```

获得特征函数

与ID3, C4.5不同, CART需要为每个特征的每个值都计算一次基尼系数, 因此计算特征的最佳分割值时得到的基尼系数都需要用于比较, 得到全局的最佳特征及特征的值。

```
1 def getBestFeat_gini(dataSet):
2     minGiniIndex=1
3     BestFeat=None
4     BestFeatVal=None
5     giniIndex=1
6     currfeatVal=None
7     for feat in dataSet.columns[:-1]:
8         featLst=list(Counter(dataSet[feat]))
9         if(len(featLst)==1):
10             pass
11         else:
12             currfeatVal,giniIndex=getBestSplit(dataSet,feat)
13             if giniIndex<minGiniIndex:
14                 BestFeat=feat
15                 BestFeatVal=currfeatVal
16                 minGiniIndex=giniIndex
17     return BestFeat,BestFeatVal
18
```

上面是所有的涉及到公式的函数。

决策树数据结构

由于决策树有多种算法, 为了能在使用同个接口调用不同算法, 数据结构需要保存算法类型 (method), 成员中必要的是child, feature, result, 由于CART算法使用了二叉树结构, 其分支推导需要一个值作为辅助, 因此有CART_val。child将被赋值为字典类型, 由于本次实验没有连续型特征, 分类时决策树可以通过样本的标签直接索引到分支上。

```
1 class decisionnode:
2     def
3     __init__(self,child,feature=None,result=None,method=None,CART_val=None):
4         self.feature=feature      #当前节点对应的特征
5         self.result=result        #当前的结果, 当child为空时有效
6         self.child=child          #存放分支
7         self.CART_val=CART_val    #CART时用于进入分支
8         self.method=method        #确定特征时使用的方法
```

构建决策树

决策树以递归形式构建, 终止条件共有三种 (算法原理中已描述), 这三种条件可以直接返回决策树节点, 其他情况需要首先按照算法分成两类, ID3, C4.5找到最佳特征后就可以通过标签直接递归到子树, CART需要找出特征的分界特征结果, 将其作为节点的标签值用于分类, 每个特征取值都会用来产生分支, 因此树的高度较大。注意生成子树时, 被分割的样本中取节点的特征取值的部分的该特征可以删去, 也可以保留, 因为他的基尼系数不会比其他分割方式低, 但是最终各个特征都只有一种取值的样本集合需要直接处理成叶节点, 删去该特征需要时间开销, 因此我没有删去。

该函数中寻找特征的函数分为两种, 一种处理ID3, C4.5, 另一种处理CART。

```
1 def CreateTree(dataSet,featDict,method='C4.5',parent_res=-1):
2     child={}
```

```

3  # 数据集为空，等于上一步没有分
4      if dataSet.empty:
5          return decisionnode(result=parent_res, feature=None, child=None)
6      resList=dataSet.loc[:, 'Label']
7      labelCnt=Counter(resList)
8      parent_res=max(labelCnt, key=labelCnt.get)
9  # 只有一种结果，已完全分开
10     if len(labelCnt)==1:
11         return
12     decisionnode(result=resList.values[0], feature=None, child=None)
13 # 没有可以分的特征了
14     if dataSet.shape[1]==1:
15         return decisionnode(result=parent_res, feature=None, child=None)
16     bestVal=None
17     if method=='CART':
18         bestfeat,bestVal=getBestFeat_gini(dataSet)
19         bestfeat,bestVal=getBestFeat_gini(dataSet)
20         if(bestfeat==None):
21             return decisionnode(result=0, feature=None, child=None)
22         else:
23             child[True]=CreateTree(dataSet[dataSet[bestfeat]==bestVal].drop(bestfeat, axis=1), featDict, method, parent_res)
24             child[False]=CreateTree(dataSet[dataSet[bestfeat]!=bestVal], featDict, method, parent_res)
25         else:
26             bestfeat=getBestFeature(dataSet, method)
27             for label in featDict[bestfeat]:
28                 child[label]=CreateTree(dataSet[dataSet[bestfeat]==label].drop(bestfeat, axis=1), featDict, method, parent_res)
29 # 返回时设置使用的方法
30     return
31     decisionnode(child=child, feature=bestfeat, result=parent_res, method=method, CART_val=bestVal)

```

收集特征取值

最初的做法中，我只在构建决策树时收集特征取值，但是训练时发现有些特征的取值会不在集合，导致分类时样本找不到对应的分支出错。因此特征取值需要在整个数据集生成时统计。返回的结果是主键为特征，值为特征取值的字典。

```

1  def collect_feat(dataSet):
2      featDict={}
3      for feat in dataSet.columns[:-1]:
4          featDict[feat]=list(Counter(dataSet[feat]))
5      return featDict

```

分类

分类是递归的过程，对于ID3，C4.5，使用测试目标的当前节点特征对应的值作为主键在字典中索引即可找到下一个决策树节点，当分支不存在时，即可返回结果。

```

1 def classify(tree,target,featIndex=None):
2     if tree.child!=None:
3         return
4     classify(tree.child[target[featIndex[tree.feature]]],target,featIndex)
5     else:
6         return tree.result

```

对于CART，只需要判断测试目标的当前节点特征对应的值和决策树节点的值是否相等，即可找到下一个分支，同样的，分支不存在时，返回结果。

```

1 def classify_CART(tree,target,featIndex):
2     if type(target)==pd.core.series.Series:
3         if tree.child!=None:
4             return
5     classify_CART(tree.child[target[tree.feature]==tree.CART_val],target,featIndex)
6     else:
7         return tree.result
8     else:
9         if tree.child!=None:
10            return
11    classify_CART(tree.child[target[featIndex[tree.feature]]==tree.CART_val],target,featIndex)
12    else:
13        return tree.result

```

验证

验证函数需要取出每一行作为样本，使用分类函数得到结果，并与样本的标签做比较，统计正确率并返回。此处需要根据使用的方法使用不同的分类函数，方法名通过树的成员获得。

```

1 def validate(tree,v_set):
2     # 记录列索引对应的位置
3     featIndex={}
4     i=0
5     for feat in v_set.columns:
6         featIndex[feat]=i
7         i+=1
8     corrCnt=0
9     if tree.method=='CART':
10        for row in v_set.iterrows():
11            if classify_CART(tree,list(row[1]),featIndex)==v_set['Label']
12            [row[0]]:
13                corrCnt+=1
14        # ID3/C4.5
15        else:
16            for row in v_set.iterrows():
17                if classify(tree,list(row[1]),featIndex)==v_set['Label']
18                [row[0]]:
19                    corrCnt+=1
20    return corrCnt/v_set.shape[0]

```

训练模块

这次实验开始需要自己分割数据集，我使用了k-fold方法，即将数据集分为k份，用其中k-1份来训练得到模型，剩下一份用于验证，模型评估通过计算k次准确率，取均值得到。也就是说训练时会生成k个决策树。

k-fold

将数据集分为k份，只需计算区间起点和终点，取出区间中的数据作为验证集，剩余的作为训练集，将两个集合即可。

```
1 def k_fold(dataSet,k,i):
2     avg_size=dataSet.shape[0]/k
3     begin=int(avg_size*(i))
4     end=int(avg_size*(i+1))
5     train_set=dataSet.drop(labels=range(begin,end),axis=0)
6     return train_set, dataSet.loc[begin:end-1]
```

训练函数

将输入的数据集分解成训练集和测试集，构建决策树，计算准确率并输出即可。

```
1 def train(dataSet,k,method='C4.5'):
2     accuracy=0.0
3     for i in range(k):
4         s1,s2=k_fold(dataSet,k,i)
5         tree=CreateTree(s1,collect_feat(dataSet),method)
6         accuracy+=validate(tree,s2)
7         print(i," finished")
8     return accuracy/k
```

主模块

主模块将获取命令行参数作为k的最大值，对每个方法遍历k从2到最大值进行训练。训练后进行准确率的测试，并将得到的准确率记录到字典中最终保存在本地的csv文件里。

```
1 import pandas as pd
2
3 def main():
4     res={}
5     ds=pd.read_csv('lab2_dataset/car_train.csv')
6     # 命令行参数
7     k = sys.argv[1]
8     methods = ['ID3', 'C4.5', 'CART']
9     # 为每个方法训练决策树
10    for method in methods:
11        res[method]=[]
12    # 为每个k训练决策树
13    for i in range(2,int(k)):
14        print("k=", i, ":running...\b")
15    # 记录准确率并加入字典中的列表
16    acc=train(ds, int(i), method)
17    res[method].append(acc)
18    print(method,"'s accuracy",acc)
19    data_df = pd.DataFrame(res)
```

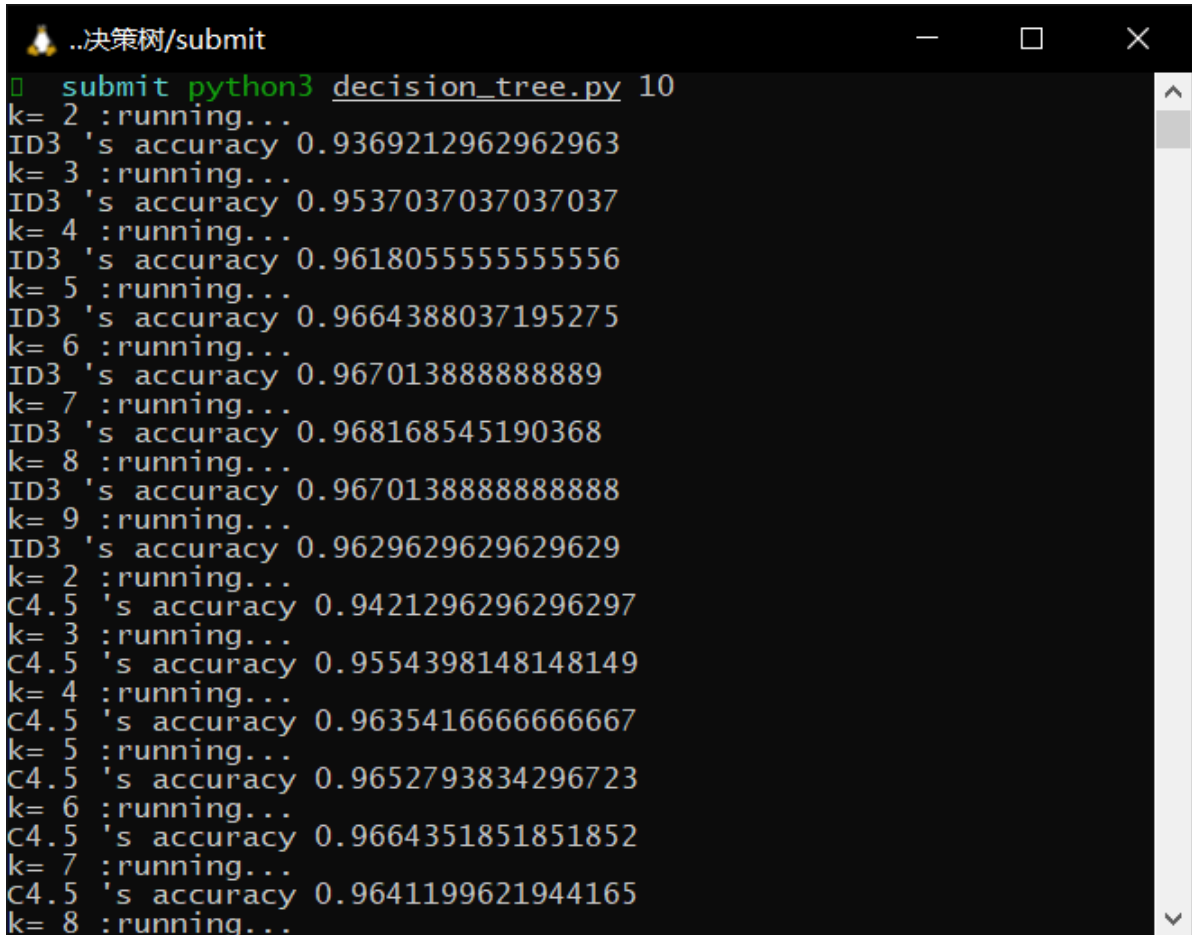


```
20 # 保存在当前目录下的csv文件中
21 data_df.to_csv('result.csv')
```

实验结果以及分析

本次实验不考虑数据集划分，则没有参数，给定的训练集和算法可以得到唯一的决策树。由于我使用了k-fold方法，调整k的大小可以得到不同的准确率结果。

下图是运行的部分结果。

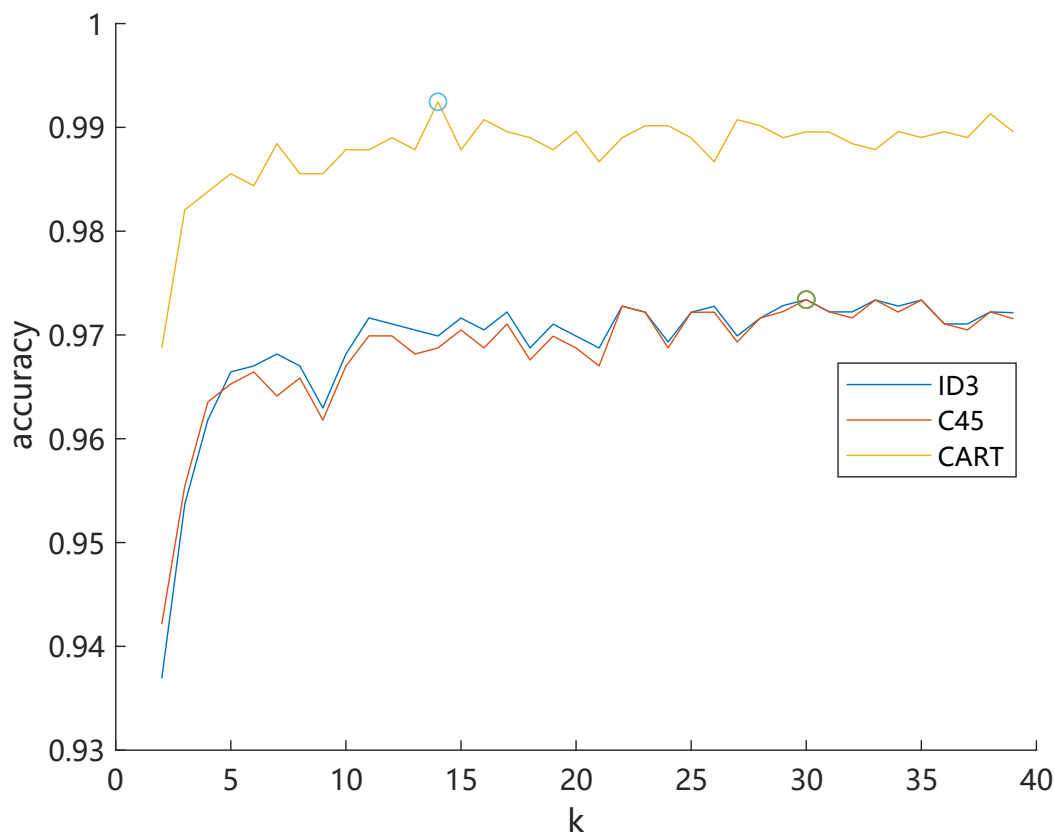


```
..决策树/submit
submit python3 decision_tree.py 10
k= 2 :running...
ID3 's accuracy 0.9369212962962963
k= 3 :running...
ID3 's accuracy 0.9537037037037037
k= 4 :running...
ID3 's accuracy 0.9618055555555556
k= 5 :running...
ID3 's accuracy 0.9664388037195275
k= 6 :running...
ID3 's accuracy 0.9670138888888889
k= 7 :running...
ID3 's accuracy 0.968168545190368
k= 8 :running...
ID3 's accuracy 0.9670138888888888
k= 9 :running...
ID3 's accuracy 0.9629629629629629
k= 2 :running...
C4.5 's accuracy 0.9421296296296297
k= 3 :running...
C4.5 's accuracy 0.9554398148148149
k= 4 :running...
C4.5 's accuracy 0.9635416666666667
k= 5 :running...
C4.5 's accuracy 0.9652793834296723
k= 6 :running...
C4.5 's accuracy 0.9664351851851852
k= 7 :running...
C4.5 's accuracy 0.9641199621944165
k= 8 :running...
```

保存在本地的csv文件中的部分内容如下（取 $k < 20$ ）：

k	ID3	C4.5	CART
2	0.936921296	0.94212963	0.96875
3	0.953703704	0.955439815	0.9820601851851851
4	0.961805556	0.963541667	0.9837962962962963
5	0.966438804	0.965279383	0.9855374047080506
6	0.967013889	0.966435185	0.984375
7	0.968168545	0.964119962	0.9884279178245802
8	0.967013889	0.965856481	0.9855324074074076
9	0.962962963	0.961805556	0.9855324074074076
10	0.968157682	0.967001613	0.9878444683425192
11	0.971641758	0.969911971	0.9878401852924146
12	0.971064815	0.969907407	0.9890046296296297
13	0.970476892	0.968163416	0.9878498694288166
14	0.969900903	0.968748829	0.9924646884717695
15	0.971629185	0.97047976	0.9878410794602698
16	0.970486111	0.96875	0.9907407407407407
17	0.972215561	0.971062158	0.9895908275948704
18	0.96875	0.967592593	0.9890046296296295
19	0.971042992	0.969886254	0.9878478246899299
max k	17	17	14
max accuracy	0.972215561	0.971062158	0.992464688471770

ID3和C4.5都始终保持93%以上的准确率，并且大部分情况在96%以上，而CART则更高，几乎达到98%。可见在选择特征时，基尼系数是一个更好的标准。



使用表格中的信息作图后可以发现，以 $k=9$ 为分界线，三种方法都有明显的从下降趋势变为上升趋势，超过16后有一定下降，我认为这是因为 $k=10$ 之前的决策树是欠拟合的。 $k=10$ 后训练集的样本数量较多，因此决策树接近收敛了。而 $k=16$ 之后，准确率在有明显下降的同时，开始出现较大波动，这可能是因为训练集较大而过拟合，波动则可能是因为验证集的样本个数下降，导致结果的随机性增加。 **k 较大时的准确率没有什么意义。**不过可以注意到，虽然三个方法的准确率并不相同，变化的趋势却基本相同。ID3和C4.5甚至在训练集较大时结果基本一样，原因是该训练集的特征取值并不多，两种算法在训练样本充分时没有很大的差别。

此外，由于本次实验使用了k-fold方法，训练的时间并不短。运行时可以发现CART的运行速度较快。

思考题

决策树有哪些避免过拟合的方法？

1. 进行预剪枝或后剪枝
 - 预剪枝
 - 在决策树生成过程中进行，若产生分支不会提高验证集上的准确率，则不产生分支
 - 后剪枝
 - 在决策树生成过程中进行，若把非叶节点变为叶节点不降低在验证集上的准确率，则变成叶节点。
2. 对原训练集做预处理，筛选出能够更好反映样本特征的集合。
3. 使用随机森林算法，用随机抽样的数据生成多棵决策树，使用每棵树对样本进行分类，按照多数投票决定最终分类结果。

C4.5相比于ID3的优点是什么，C4.5又可能有什么缺点？

优点

- C4.5解决了ID3一般会优先选择有较多属性值的特征的问题。
- 可以处理连续特征

缺点

- C4.5使用的仍然是多叉树，树的规模较大。
- C4.5只能用于分类
- 信息增益率的计算公式需要更多的对数运算，训练速度更慢。

如何用决策树来进行特征选择（判断特征的重要性）？

- ID3,C4.5，使用信息增益，信息增益（率）高的特征更重要。
- CART，基尼系数最小的特征分割点对应的特征最重要。
- 选择特征构建成树后，比较修改特征后准确率的变化，准确率提高则修改后的特征更重要。要尽量避免选择分支过多的特征，比如ID。