云南大学软件学院
实 验 报 告

姓名：王子陈 学号：20171050008 专业:电子科学与技术 日期：2019/10/23 成绩：_____
任课教师：谢仲文

数据挖掘技术 实验二

一、实验目的

1. 掌握决策树分类的 ID3 算法、C4.5 算法和 CART 算法。
2. 选择一种编程语言实现 C4.5 算法。

二、实验内容

1．在一个简单的、虚拟的数据集（训练集）上应用 C4.5 算法。该数据集如下：

	天气	温度	湿度	刮风	适合运动
1	sunny	85	85	FALSE	no
2	sunny	80	90	TRUE	no
3	overcast	83	86	FALSE	yes
4	rainy	70	96	FALSE	yes
5	rainy	68	80	FALSE	yes
6	rainy	65	70	TRUE	no
7	overcast	64	65	TRUE	yes
8	sunny	72	95	FALSE	no
9	sunny	69	70	FALSE	yes
10	rainy	75	80	FALSE	yes
11	sunny	75	70	TRUE	yes
12	overcast	72	90	TRUE	yes
13	overcast	81	75	FALSE	yes
14	rainy	71	91	TRUE	no

2．构建预测数据集，并在数据集上应用分类模型。
3. 简要对比 ID3 算法、C4.5 算法和 CART 算法。

三、实验要求

1. 完成实验内容，源码作为实验报告附件一起打为一个压缩包提供。该压缩包要包含实验报告、代码文件。
2. 关键部分要求有注释，注释量不低于 20%
3. 要求独立完成，不得抄袭代码。

四、关键实验步骤（请粘贴关键步骤、代码、实验结果）

(使用 python 语言)

#计算信息熵

```
def cal_Ent(data_set):
    num=len(data_set) #记录条数
    label_counts={}   #字典放类别(yes/no)
    for i in range(num):   #遍历 data_set 中的每一条记录
        current_label=data_set[i][-1] #只取最后一列数据(Play?)
        if current_label not in label_counts.keys(): #如果这个属性没在 label_counts 中出现
            label_counts[current_label]=0   #就把这个键的值赋为 0
        label_counts[current_label]+=1   #让当前键的值+1
    Ent=0.0
    for key in label_counts:
        prob=float(label_counts[key])/num
        Ent=Ent-prob*log(prob,2)
    return Ent
```

```
# In[2]:
```

#离散属性,切割数据集,返回不包含 axis 列的子数据集

```
def split_data_set(data_set, axis, value):
    ret_data_set = []
    for feat_vec in data_set:
        if feat_vec[axis] == value:
            reduce_feat_vec = feat_vec[:axis] #去掉 axis 那个值
            reduce_feat_vec.extend(feat_vec[axis + 1:])
            ret_data_set.append(reduce_feat_vec)
    return ret_data_set
#split_data_set(data_set1,0,"rainy")
```

```
# In[3]:
```

#按连续属性切割数据集的方法:

```
def split_con_data_set(data_set,axis,value,LorR='L'):
    return_data_set=[]
    if LorR =='L': #构建左子树
        for feature_vec in data_set:
            if feature_vec[axis]<=value:   #属性值小于等于 value 分在左子树
                reduce_feat_vec = feature_vec[:axis]
                reduce_feat_vec.extend(feature_vec[axis + 1:])
```

```
                return_data_set.append(reduce_feat_vec)
    else: #大于 value 分在右子树
        for feature_vec in data_set:
            if feature_vec[axis]>value:
                reduce_feat_vec = feature_vec[:axis]
                reduce_feat_vec.extend(feature_vec[axis + 1:])
                return_data_set.append(reduce_feat_vec)
    return return_data_set


# In[4]:
```

#把连续属性离散化,排序以后,只有在类别发生改变的地方才需要切开,找到使信息增益率最大的分类方式

```
best_division=[0]#用这个列表存分界线,初始值为 0
def cal_continue_gain_ratio (data_set_raw,attribute):
    data_set=sorted(data_set_raw,key=operator.itemgetter(labels.index(attribute)),reverse=False)#先排序
    num=len(data_set)
    min_entropy=cal_Ent(data_set) #初始为未分类的信息熵
    left_sub_data_set=[]
    right_sub_data_set=[]
    for i in range(num-1):#从第一条记录遍历到倒数第二条,因为下面有 i+1
        if data_set[i][-1]!= data_set[i+1][-1]:#如果上下两条的类别不同,就划分一次
            for j in range(num):
                if j<=i:   #把在分界线之上的记录,分在左子树
                    left_sub_data_set.append(data_set[j])
                else:        #在分界线之下的就分在右子树
                    right_sub_data_set.append(data_set[j])
            #对这次的分类结果计算条件熵
            prob_L=len(left_sub_data_set)/num
            prob_R=len(right_sub_data_set)/num
            sigma_ent=(prob_L)*cal_Ent(left_sub_data_set)+(prob_R)*cal_Ent(right_sub_data_set)
            if sigma_ent<min_entropy:
                min_entropy=sigma_ent
                best_division[0]=i #最佳分界线
            left_sub_data_set.clear()
            right_sub_data_set.clear()
    max_gain=cal_Ent(data_set)-min_entropy #计算最大信息增益
    #计算分类信息度量 H
    H=0.0
```

```
    prob_l=(best_division[0]+1)/num
    prob_r=1-prob_l
    H=-prob_l*log(prob_l,2)-prob_r*log(prob_r,2)
    #计算信息增益率
    gain_ratio=max_gain/H
    return gain_ratio


# In[5]:
```

#计算离散属性的 gain_ratio(天气/刮风)

```
def cal_discrete_gain_ratio(data_set,attribute):
    base_entropy=cal_Ent(data_set) #未分类时的信息熵
    info_gain=0   #信息增益初始为 0
    num_records=len(data_set) #父节点的记录条数
    feature_val_list=[]#用来存储每一条记录的第 label_idx 个属性的特征值
    for i in range(num_records):#遍历每一条记录
        feature_val_list.append(data_set[i][labels.index(attribute)])
    unique_feature_val_list=set(feature_val_list) #得到所有的离散特征
    #计算不同类特征的信息熵,再乘上权重求和就是条件熵
    sigma=0
    H=0
    for feature_val in unique_feature_val_list:   #遍历每一种特征
        sub_data_set=split_data_set(data_set,labels.index(attribute),feature_val) #去掉该特征值,得到以该特征
  分类的子树
        prob=len(sub_data_set)/num_records
        sigma+=prob*cal_Ent(sub_data_set)
        #计算分裂度量信息度量 H
        H=H-prob*log(prob,2)

    info_gain=base_entropy-sigma

    #计算信息增益率
    gain_ratio=info_gain/H
    return gain_ratio


# In[6]:
```

#找到最能有效分类的属性

```
def choose_best_attribute_to_split(data_set,labels):
```

```
    if len(labels)==1:   #如果只剩 1 个属性了,最佳属性就是它
        return labels[0]
    best_gain_ratio=0
    best_attribute=labels[-1]
    for attribute in labels:
        if attribute=='weather' or attribute=='windy':
            gain_ratio=cal_discrete_gain_ratio(data_set,attribute)
        else:
            gain_ratio=cal_continue_gain_ratio(data_set,attribute)

        if gain_ratio>best_gain_ratio:
                best_gain_ratio=gain_ratio
                best_attribute=attribute
    return best_attribute


# In[7]:
#统计 yes 和 no 出现的次数,并按大小排序,返回出现次数多的类别
def majority_count(class_list):
    class_count={}   #类别统计
    for vote in class_list:   #yes 和 no 两种
        if vote not in class_count.keys():
            class_count[vote]=0
        class_count[vote]+=1
    sorted_class_count=sorted(class_count.items(),key=operator.itemgetter(1))
    return sorted_class_count[-1][0]


# In[8]:
#构建决策树
def create_tree(data_set,labels):
    class_list=[records[-1] for records in data_set]   #只取类别:yes/no
    if class_list.count(class_list[0])==len(class_list):#最后一个元素的个数等于列表长度,说明很"纯",里面只
                                                        有一种类别
        return class_list[0]   #就可以返回分类结果:yes / no
    if len(data_set[0])==1:    #如果所有属性都被遍历完了,
        return majority_count(class_list) #返回出现次数最多的类别

    #按照信息增益率最大选取分类特征属性
```

```
    best_attribute=choose_best_attribute_to_split(data_set,labels)#返回最优划分属性

    #以最优属性开始划分:
    #如果是离散属性(天气和刮风),庶出的每一颗子树不包含最优属性:
    if best_attribute=='weather'or best_attribute=='windy':
        best_attribute_label=best_attribute     #字典的 key 值
        decision_tree={best_attribute_label:{}} #构建树的字典

        feature_values=[example[labels.index(best_attribute)]for example in data_set]#把最优属性的一列取出
        unique_feature_values=set(feature_values) #取出互不重复的特征值集合
        #print(unique_feature_values)
        axis=labels.index(best_attribute)
        del(labels[labels.index(best_attribute)]) #从 labels[ ]中删除该 label,已经选择过的属性不再参与分类
        sub_labels=labels[:] #浅拷贝
        for feature_value in unique_feature_values:#遍历每一种特征
            sub_data_set=split_data_set(data_set,axis,feature_value)#得到每个特征划分的子树
            decision_tree[best_attribute_label][feature_value]=create_tree(sub_data_set,sub_labels)#每个节点
下面再分树
            sub_labels=labels[:]
    #如果是连续属性,,分别构建左子树和右子树
    else:
        best_attribute_label=best_attribute+'<='+str(data_set[best_division[0]][labels.index(best_attribute)])
        decision_tree={best_attribute_label:{}} #构建树的字典
        axis=labels.index(best_attribute) #找到划分轴

  left_sub_data_set=split_con_data_set(data_set,axis,data_set[best_division[0]][labels.index(best_attribute)],'L')
  right_sub_data_set=split_con_data_set(data_set,axis,data_set[best_division[0]][labels.index(best_attribute)],'R')
        del(labels[axis])
        sub_labels=labels[:]
        decision_tree[best_attribute_label]['yes']=create_tree(left_sub_data_set,sub_labels)
        decision_tree[best_attribute_label]['no']=create_tree(right_sub_data_set,sub_labels)
    return decision_tree
```

代入数据集:

```
data_set=[['overcast',64,65,'TRUE','yes'],['rainy',65,70,'TRUE','no'],
          ['rainy',68,80,'FALSE','yes'],['sunny',69,70,'FALSE','yes'],
          ['rainy',70,96,'FALSE','yes'],['rainy',71,91,'TRUE','no'],
          ['sunny',72,95,'FALSE','no'],['overcast',72,90,'TRUE','yes'],
          ['rainy',75,80,'FALSE','yes'],['sunny',75,70,'TRUE','yes'],
          ['sunny',80,90,'TRUE','no'],['overcast',81,75,'FALSE','yes'],
```

['overcast',83,86,'FALSE','yes'],['sunny',85,85,'FALSE','no']]

labels=['weather','temperature','humidity','windy']

mytree=create_tree(data_set,labels)

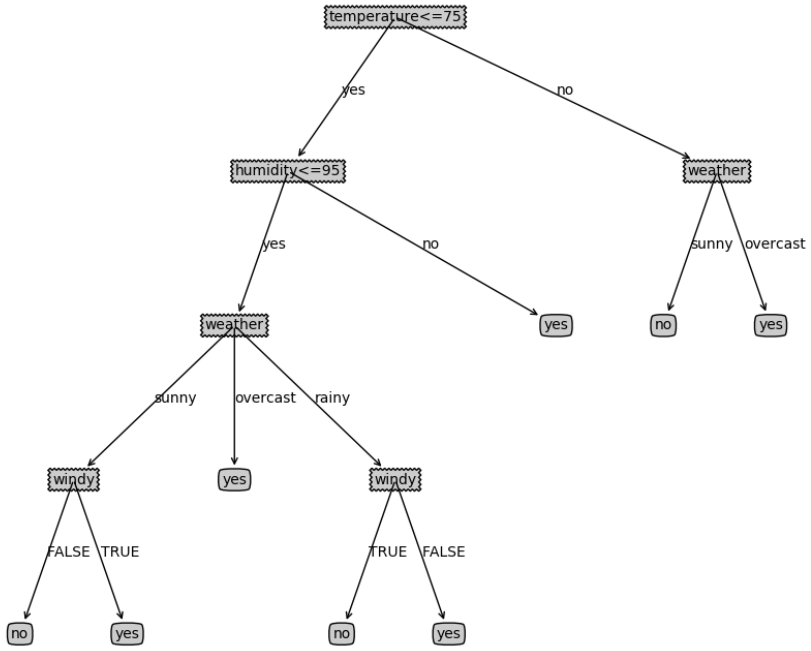
print(mytree)

create_plot(mytree)

运行结果:是一个字典

{'temperature<=75': {'yes': {'humidity<=95': {'yes': {'weather': {'sunny': {'windy': {'FALSE': 'no', 'TRUE': 'yes'}}, 'rainy': {'windy': {'TRUE': 'no', 'FALSE': 'yes'}}, 'overcast': 'yes'}}, 'no': 'yes'}}, 'no': {'weather': {'sunny': 'no', 'overcast': 'yes'}}}}

字典形式的决策树仍然不易理解，利用 Matplotlib 库的 annotate（注释）模块绘制决策树，就可以很直观的看出决策树的结构。



测试决策树:

用原数据集重新放入决策树,进行分类,得到的类别与实际值对比:(有一条记录不同)

实际值	yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no
测试分类	yes	no	yes	no	yes	no	no	yes	yes	yes	no	yes	yes	no

由此看出,得到的决策树的准确率为 71.4%

测试分类部分的代码:

```
def classify(inputTree,featLabels,testVec):
    # 得到树中的第一个特征
    firstStr = list(inputTree.keys())[0]
    # 得到第一个对应的值
```

```
secondDict = inputTree[firstStr]
if firstStr.find('<')==-1:
    firstLabel=firstStr
else:
    firstLabel=firstStr[:firstStr.find('<')]
# 得到树中第一个特征对应的索引
featIndex = featLabels.index(firstLabel)
classLabel=None
# 遍历树
for key in secondDict.keys():
    if featLabels[featIndex]=='weather'or featLabels[featIndex]=='windy':
        # 如果在 secondDict[key]中找到了 testVec[featIndex]
        if testVec[featIndex] == key:
            # 判断 secondDict[key]是否为字典
            if type(secondDict[key]).__name__ == 'dict':
                # 若为字典，递归的寻找 testVec
                classLabel = classify(secondDict[key], featLabels, testVec)
            else:
                # 若 secondDict[key]为标签值，则将 secondDict[key]赋给 classLabel
                classLabel = secondDict[key]
    else: #连续属性：温度和湿度
        splitValue = eval(firstStr[firstStr.find('=')+1:])
        if testVec[featIndex]<=splitValue: #进入左子树
            if type(secondDict['yes']).__name__=='dict': #该分支不是叶子节点,递归
                classLabel=classify(secondDict['yes'],featLabels,testVec)
            else: #如果是叶子,返回结果
                classLabel=secondDict['yes']
        else:
            if type(secondDict['no']).__name__=='dict':
                classLabel=classify(secondDict['no'],featLabels,testVec)
            else:#如果是叶子节点,返回结果
                classLabel=secondDict['no']
# 返回类标签
return classLabel
```

简要对比 ID3 算法、C4.5 算法和 CART 算法。

ID3 算法是在决策树各个节点上应用信息增益准则选择特征递归地构建决策树。信息增益大, 说明使用该特征后划分得到的子集纯度越高，即不确定性越小。**缺点：**信息增益偏向取值较多的特征（原因：当特征的取值较多时，根据此特征划分更容易得到纯度更高的子集，因此划分后的熵更低，即不确定性更低，因此信息增益更大）

C4.5 算法是对 ID3 算法做了改进，在生成决策树过程中采用信息增益比来选择特征。信息增益会偏向取值较多的特征，使用信息增益比可以对这一问题进行校正。能够完成对连续属性的离散化处理；能够对不完整数据进行处理。

基尼指数 Gini(D)表示集合 D 的不确定性，基尼指数 Gini(D,A)表示经 A 分割后集合 D 的不确定性。基尼指数值越大，样本集合的不确定性也就越大, 选 Gini 指数最小的特征作为分裂标准。CART 和 C4.5 支持数据特征为连续分布时的处理，主要通过使用二元切分来处理连续型变量，即求一个特定的值——分裂值：特征值小于等于分裂值就走左子树，或者就走右子树。