实验七　分类 决策树算法

【实验目的】

1. 了解 Anaconda 和 python 的使用与设置。
2. 掌握分类决策树算法的基本原理和实现方法。

【实验学时】

建议 2 学时

【实验环境配置】

1、 Windows 环境
2、 Anaconda
3、 Pandas

【实验原理】

数据挖掘中分类决策树算法的实现：

1、Pandas 数据导入
2、决策树算法
3、sk-learn 库的决策树模块使用

【实验步骤】

1. 打开 Anaconda 的 jupyter notebook，创建实验七，明确标题和步骤。

2. 导入 pandas 库和 os 库，从 lenses 数据文件中，导入实验训练数据和测试数据。
（注：可能涉及的函数 os.path.join, os.walk, pandas.read_csv）

```python
from math import log

def calcShannonEnt(dataSet):  # 计算数据集的香农熵
    numEntries = len(dataSet)  # 数据集中实例的总数
    labelCounts = {}  # 字典键值记录当前类别出现的次数
    for featVec in dataSet:
        currentLabel = featVec[-1]  # 当前标签为字典
        if currentLabel not in labelCounts.keys():
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries  # 计算每个类别的概率
        shannonEnt -= prob * log(prob, 2)  # 熵值
    return shannonEnt

def createDataSet():
    dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
    labels = ['no surfacing', 'flippers']
    return dataSet, labels

myDat, lables = createDataSet()
myDat
```

```
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
```

```python
calcShannonEnt(myDat)
```

```
0.9709505944546686
```

3. 对导入的实验数据，根据《机器学习实战》教程第三章，完成分类决策树 ID3 算法，并完成对于 lenses 的分类识别。

（1）按照给定特征分割数据集（待划分数据集,划分数据集的特征，需要返回的特征的值）

```python
def splitDataSet(dataSet, axis, value):  # 按照给定特征分割数据集（待划分数据集,划分数据集的特征，需要返回的特征的值）
    retDataSet = []  # 新的list对象
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]  # 接收axis之前的特征
            reducedFeatVec.extend(featVec[axis+1:])  # 将axis之后的特征进行扩展
            retDataSet.append(reducedFeatVec)
    return retDataSet  # 得到依据该特征划分后的数据子集

splitDataSet(myDat, 0, 1)
```

```
[[1, 'yes'], [1, 'yes'], [0, 'no']]
```

（2）选取最好的数据集的划分方式(选择最好的 feature 来划分数据集）

```python
def chooseBeatFeatureToSplit(dataSet):  # 选取最好的数据集的划分方式
    numFeatures = len(dataSet[0]) - 1  # 特征数量
    baseEntropy = calcShannonEnt(dataSet)  # 计算数据集的香农熵
    bestInfoGain = 0.0  # 初始化最优的信息增益
    bestFeature = -1
    for i in range(numFeatures):  # 依据特征循环
        featList = [example[i] for example in dataSet]  # 第i个特征的所有内容
        uniqueVals = set(featList)  # 去重
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)  # 数据子集
            prob = len(subDataSet)/float(len(dataSet))  # 概率
            newEntropy += prob * calcShannonEnt(subDataSet)  # 计算数据子集中每个特征的香农熵
        infoGain = baseEntropy - newEntropy  # 信息增益
        if infoGain > bestInfoGain:  # 比较当前信息增益和上一次的
            bestInfoGain = infoGain
            bestFeature = i
    return bestFeature

chooseBeatFeatureToSplit(myDat)
```

```
0
```

（4）构建决策树

```python
def majorityCnt(classList):  # 分类次数最多的一类，参数（分类名称的列表）
    classCount = {}  # 空字典
    for vote in classList:  # 创建键值为classList中唯一的数据字典
        if vote in classCount.keys():  # 统计每个类别的出现的频率
            classCount[vote] += 0
        classCount[vote] += 1
    sortedclassCount = sorted(classCount.items(), key=operator.itemgetter(1), reverse=True)  # 根据频率排序
    return sortedclassCount[0][0]  # 返回出现次数最多的分类名称

def createTree(dataSet, labels):  # 创建树，参数（数据集和包含所有特征的标签列表）
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):  # 类别完全相同则停止继续划分
        return classList[0]
    if len(dataSet[0]) == 1:  # 遍历完所有特征返回出现次数最多的
        return majorityCnt(classList)
    bestFeat = chooseBeatFeatureToSplit(dataSet)  # 选取最好的特征
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel: {}}  # 字典myTree存储了树的所有信息
    del(labels[bestFeat])  # 删除已选的的特征
    featValues = [example[bestFeat] for example in dataSet]  # 得到所有属性值
    uniqueVals = set(featValues)  # 去重
    for value in uniqueVals:
        subLabels = labels[:]  # 特征子集
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value), subLabels)
    return myTree

myDat, labels = createDataSet()
createTree(myDat, labels)
```

```
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

（5）绘图函数

```
def plotNode(nodeTxt, centerPt, parentPt, nodeType):  # 定义了实际的绘图功能
    createplot.ax1.annotate(nodeTxt, xy=parentPt, xycoords='axes fraction',
                            xytext=centerPt, textcoords='axes fraction',
                            va="center", ha="center", bbox=nodeType,
                            arrowprops=arrow_args)  # 文本内容，箭头尖端，文本位置，coords分别指定坐标系

def createplot():  # 绘制总体的图
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    createplot.ax1 = plt.subplot(111, frameon=False)
    plotNode('决策节点', (0.5, 0.1), (0.1, 0.5), decisionNode)
    plotNode('叶节点', (0.8, 0.1), (0.3, 0.8), leafNode)
    plt.show()

def getNumLeafs(myTree):  # 获取叶节点的数目
    numLeafs = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':  # 测试节点的数据是否为字典
            numLeafs += getNumLeafs(secondDict[key])  # 如果是依据子树继续数
        else:
            numLeafs += 1
    return numLeafs

def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':  # 测试节点的数据是否为字典
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth:
            maxDepth = thisDepth
    return maxDepth

def retrieveTree(i):  # 输出预先存储的树的信息,用于测试
    listOfTrees = [{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}},
                   {'no surfacing': {0: 'no', 1: {'flippers': {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}]
    return listOfTrees[i]

myTree = retrieveTree(1)
myTree = retrieveTree(0)
myTree
```

```
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

```
getNumLeafs(myTree)
```

```
3
```

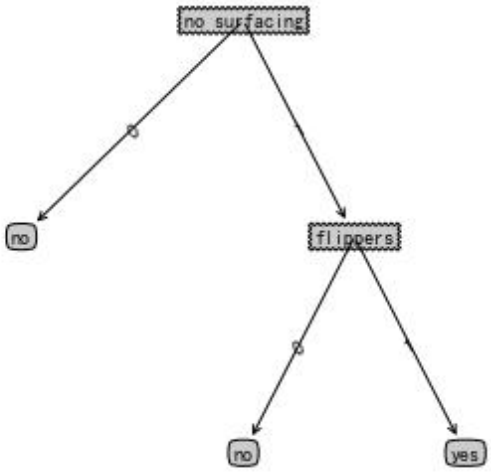
（6）对初始数据使用绘图函数进行测试

```
def plotMidText(cntrPt, parentPt, txtString):  # 在父子节点间填充文本信息
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]  # 填充信息的x横坐标
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]  # 填充信息的y纵坐标
    createplot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)

def plotTree(myTree, parentPt, nodeTxt):
    numLeafs = getNumLeafs(myTree)  # 计算树叶节点的数量
    depth = getTreeDepth(myTree)  # 计算树深
    firstStr = list(myTree.keys())[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)  # 文本位置
    plotMidText(cntrPt, parentPt, nodeTxt)  # 填充父子节点之间的信息
    plotNode(firstStr, cntrPt, parentPt, decisionNode)  # 实际绘图
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD  # 减少y的偏移
    for key in secondDict.keys():  # 遍历整棵树
        if type(secondDict[key]).__name__ == 'dict':
            plotTree(secondDict[key], cntrPt, str(key))
        else:
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD

def createplot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()  # 清除去上图数据
    axprops = dict(xticks=[], yticks=[])
    createplot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plotTree.totalW = float(getNumLeafs(inTree))  # 树宽
    plotTree.totalD = float(getTreeDepth(inTree))  # 树深
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5, 1.0), '')
    plt.show()


myTree = retrieveTree(0)
createplot(myTree)
```



完成分类函数

```
:
def classify(inputTree, featLables, testVec):
    firstStr = list(inputTree.keys())[0]
    secondList = inputTree[firstStr]
    featIndex = featLables.index(firstStr)  # 将标签字符串转换为索引
    for key in secondList.keys():
        if testVec[featIndex] == key:
            if type(secondList[key]).__name__ == 'dict':
                classLabel = classify(secondList[key], featLables, testVec)
            else:
                classLabel = secondList[key]
    return classLabel


myDat, labels = createDataSet()
myTree = retrieveTree(0)
classify(myTree, labels, [1, 0])
```

```
: 'no'
```

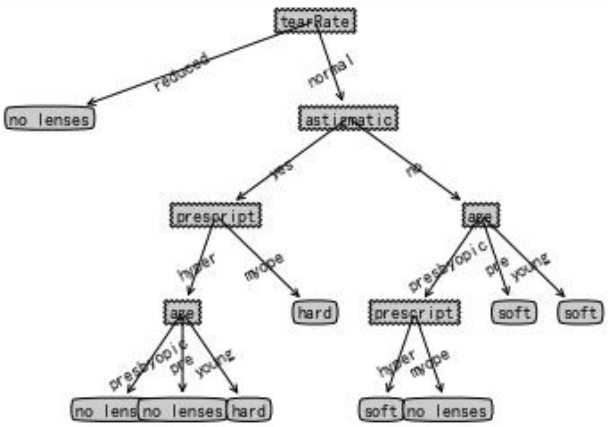
储存树函数

```
def storeTree(inputTree, filename):
    import pickle
    fw = open(filename, 'wb')
    pickle.dump(inputTree, fw)
    fw.close()
```

```
def grabTree(filename):
    import pickle
    fr = open(filename, 'rb')
    return pickle.load(fr)
```

从文件读取数据并完成决策树分类算法

```
fr = open('lenses.txt')
lenses = [inst.strip().split('\t') for inst in fr.readlines()]
lensesLabels = ['age', 'prescript', 'astigmatic', 'tearRate']
lensesTree = createTree(lenses, lensesLabels)
createplot(lensesTree)
```



4. 对导入的实验数据，根据 sklearn 的分类决策树算法模块 sklearn.tree.DecisionTreeClassifier，完成对于 lenses 数据的分类识别。

（1）数据集：

young	myope	no	reduced	no lenses
young	myope	no	normal	soft
young	myope	yes	reduced	no lenses
young	myope	yes	normal	hard
young	hyper	no	reduced	no lenses
young	hyper	no	normal	soft
young	hyper	yes	reduced	no lenses
young	hyper	yes	normal	hard
pre	myope	no	reduced	no lenses
pre	myope	no	normal	soft
pre	myope	yes	reduced	no lenses
pre	myope	yes	normal	hard
pre	hyper	no	reduced	no lenses
pre	hyper	no	normal	soft
pre	hyper	yes	reduced	no lenses
pre	hyper	yes	normal	no lenses
presbyopic	myope	no	reduced	no lenses
presbyopic	myope	no	normal	no lenses
presbyopic	myope	yes	reduced	no lenses
presbyopic	myope	yes	normal	hard

（2）代码实现

导入数据

```
import os
import pandas as pd
dir_ds=os.path.join(os.path.curdir,'lenses.txt')
df_data=pd.read_csv(dir_ds,sep='\t',header=None,names=['age','spectacle','astigmatic','tear','class'])
df_data
```

```
from sklearn.model_selection import train_test_split
training_set,testing_set=train_test_split(df_data,test_size=0.2)
training_set=training_set.reset_index()
training_set
```

数据整理

```
testing_set=testing_set.reset_index()
testing_set
```

	index	age	spectacle	astigmatic	tear	class
0	2	young	myope	yes	reduced	no lenses
1	9	pre	myope	no	normal	soft
2	20	presbyopic	hyper	no	reduced	no lenses
3	21	presbyopic	hyper	no	normal	soft
4	12	pre	hyper	no	reduced	no lenses

```
from  sklearn import tree
X=df_data.iloc[:,0:4]
Y=df_data.iloc[:,4]
from sklearn.preprocessing import LabelEncoder
ds=LabelEncoder()
ds.fit(['no lenses','hard','soft','young','pre','presbyopic','myope','hyper','no','yes','reduced','normal'])
Y1=ds.transform(Y.iloc[:])
Y1
```

```
array([4, 9, 4, 0, 4, 9, 4, 0, 4, 9, 4, 0, 4, 9, 4, 4, 4, 4, 4, 0, 4, 9,
       4, 4])
```

```
X1=[ds.transform(r) for i,r in X.iterrows()]
X1
```

```
[array([11,  2,  3,  8]),
 array([11,  2,  3,  5]),
 array([11,  2, 10,  8]),
 array([11,  2, 10,  5]),
 array([11,  1,  3,  8])
```

使用 sklearn 完成分类算法，并将运行结果存入文档 dt.txt

```
model_dt=tree.DecisionTreeClassifier(criterion='entropy')
model_dt=model_dt.fit(X1,Y1)
tree.export_graphviz(model_dt,out_file='dt.txt')
```

（3）运行结果

dt.txt 文档结果：

```
digraph Tree {
node [shape=box] ;
0 [label="X[3] <= 6.5\nentropy = 1.326\nsamples = 24\nvalue = [4, 15, 5]"] ;
1 [label="X[2] <= 6.5\nentropy = 1.555\nsamples = 12\nvalue = [4, 3, 5]"] ;
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;
2 [label="X[1] <= 1.5\nentropy = 0.65\nsamples = 6\nvalue = [0, 1, 5]"] ;
1 -> 2 ;
3 [label="entropy = 0.0\nsamples = 3\nvalue = [0, 0, 3]"] ;
2 -> 3 ;
4 [label="X[0] <= 6.5\nentropy = 0.918\nsamples = 3\nvalue = [0, 1, 2]"] ;
2 -> 4 ;
5 [label="entropy = 0.0\nsamples = 1\nvalue = [0, 0, 1]"] ;
4 -> 5 ;
6 [label="X[0] <= 9.0\nentropy = 1.0\nsamples = 2\nvalue = [0, 1, 1]"] ;
4 -> 6 ;
7 [label="entropy = 0.0\nsamples = 1\nvalue = [0, 1, 0]"] ;
6 -> 7 ;
8 [label="entropy = 0.0\nsamples = 1\nvalue = [0, 0, 1]"] ;
6 -> 8 ;
9 [label="X[1] <= 1.5\nentropy = 0.918\nsamples = 6\nvalue = [4, 2, 0]"] ;
1 -> 9 ;
```