人工智能

**1 什么是决策树**

决策树（Decision Tree）是一种基本的分类与回归方法，本文主要讨论分类决策树。决策树模型呈树形结构，在分类问题中，表示基于特征对数据进行分类的过程。它可以认为是if-then规则的集合。每个内部节点表示在属性上的一个测试，每个分支代表一个测试输出，每个叶节点代表一种类别。   
决策树的优点：   
1）可以自学习。在学习过程中不需要使用者了解过多的背景知识，只需要对训练数据进行较好的标注，就能进行学习。   
2）决策树模型可读性好，具有描述性，有助于人工分析；   
3）效率高，决策树只需要一次构建，就可以反复使用，每一次预测的最大计算次数不超过决策树的深度。

## 2 构建决策树

## 决策树构建的基本步骤如下：  1. 开始，所有记录看作一个节点  2. 遍历每个特征的每一种分裂方式，找到最好的分裂特征（分裂点）  3. 分裂成两个或多个节点  4. 对分裂后的节点分别继续执行2-3步，直到每个节点足够“纯”为止  如何评估分裂点的好坏？如果一个分裂点可以将当前的所有节点分为两类，使得每一类都很“纯”，也就是同一类的记录较多，那么就是一个好分裂点。  具体实践中，到底选择哪个特征作为当前分裂特征，常用的有下面三种算法：  ID3：使用信息增益g(D,A)进行特征选择  C4.5：信息增益率 =g(D,A)/H(A)  CART：基尼系数  一个特征的信息增益(或信息增益率，或基尼系数)越大，表明特征对样本的熵的减少能力更强，这个特征使得数据由不确定性到确定性的能力越强。

实现代码：

#!/usr/bin/python

#encoding:utf-8

from math import log

import operator

import treePlotter

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

def createDataSet():

dataSet = [[1, 1, 'yes'],

[1, 1, 'yes'],

[1, 0, 'no'],

[0, 1, 'no'],

[0, 1, 'no']]

labels = ['no surfacing','flippers']

#change to discrete values

return dataSet, labels

##### 计算信息熵 ######

def calcShannonEnt(dataSet):

numEntries = len(dataSet) # 样本数

labelCounts = {} # 创建一个数据字典：key是最后一列的数值（即标签，也就是目标分类的类别），value是属于该类别的样本个数

for featVec in dataSet: # 遍历整个数据集，每次取一行

currentLabel = featVec[-1] #取该行最后一列的值

if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0

labelCounts[currentLabel] += 1

shannonEnt = 0.0 # 初始化信息熵

for key in labelCounts:

prob = float(labelCounts[key])/numEntries

shannonEnt -= prob \* log(prob,2) #log base 2 计算信息熵

return shannonEnt

##### 按给定的特征划分数据 #########

def splitDataSet(dataSet, axis, value): #axis是dataSet数据集下要进行特征划分的列号例如outlook是0列，value是该列下某个特征值，0列中的sunny

retDataSet = []

for featVec in dataSet: #遍历数据集，并抽取按axis的当前value特征进划分的数据集(不包括axis列的值)

if featVec[axis] == value: #

reducedFeatVec = featVec[:axis] #chop out axis used for splitting

reducedFeatVec.extend(featVec[axis+1:])

retDataSet.append(reducedFeatVec)

# print axis,value,reducedFeatVec

# print retDataSet

return retDataSet

##### 选取当前数据集下，用于划分数据集的最优特征

def chooseBestFeatureToSplit(dataSet):

numFeatures = len(dataSet[0]) - 1 #获取当前数据集的特征个数，最后一列是分类标签

baseEntropy = calcShannonEnt(dataSet) #计算当前数据集的信息熵

bestInfoGain = 0.0; bestFeature = -1 #初始化最优信息增益和最优的特征

for i in range(numFeatures): #遍历每个特征iterate over all the features

featList = [example[i] for example in dataSet]#获取数据集中当前特征下的所有值

uniqueVals = set(featList) # 获取当前特征值，例如outlook下有sunny、overcast、rainy

newEntropy = 0.0

for value in uniqueVals: #计算每种划分方式的信息熵

subDataSet = splitDataSet(dataSet, i, value)

prob = len(subDataSet)/float(len(dataSet))

newEntropy += prob \* calcShannonEnt(subDataSet)

infoGain = baseEntropy - newEntropy #计算信息增益

if (infoGain > bestInfoGain): #比较每个特征的信息增益，只要最好的信息增益

bestInfoGain = infoGain #if better than current best, set to best

bestFeature = i

return bestFeature #returns an integer

#####该函数使用分类名称的列表，然后创建键值为classList中唯一值的数据字典。字典

#####对象的存储了classList中每个类标签出现的频率。最后利用operator操作键值排序字典，

#####并返回出现次数最多的分类名称

def majorityCnt(classList):

classCount={}

for vote in classList:

if vote not in classCount.keys(): classCount[vote] = 0

classCount[vote] += 1

sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1), reverse=True)

return sortedClassCount[0][0]

##### 生成决策树主方法

def createTree(dataSet,labels):

classList = [example[-1] for example in dataSet] # 返回当前数据集下标签列所有值

if classList.count(classList[0]) == len(classList):

return classList[0]#当类别完全相同时则停止继续划分，直接返回该类的标签

if len(dataSet[0]) == 1: ##遍历完所有的特征时，仍然不能将数据集划分成仅包含唯一类别的分组 dataSet

return majorityCnt(classList) #由于无法简单的返回唯一的类标签，这里就返回出现次数最多的类别作为返回值

bestFeat = chooseBestFeatureToSplit(dataSet) # 获取最好的分类特征索引

bestFeatLabel = labels[bestFeat] #获取该特征的名字

# 这里直接使用字典变量来存储树信息，这对于绘制树形图很重要。

myTree = {bestFeatLabel:{}} #当前数据集选取最好的特征存储在bestFeat中

del(labels[bestFeat]) #删除已经在选取的特征

featValues = [example[bestFeat] for example in dataSet]

uniqueVals = set(featValues)

for value in uniqueVals:

subLabels = labels[:] #copy all of labels, so trees don't mess up existing labels

myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),subLabels)

return myTree

def classify(inputTree,featLabels,testVec):

firstStr = inputTree.keys()[0]

secondDict = inputTree[firstStr]

featIndex = featLabels.index(firstStr)

key = testVec[featIndex]

valueOfFeat = secondDict[key]

if isinstance(valueOfFeat, dict):

classLabel = classify(valueOfFeat, featLabels, testVec)

else: classLabel = valueOfFeat

return classLabel

def storeTree(inputTree,filename):

import pickle

fw = open(filename,'w')

pickle.dump(inputTree,fw)

fw.close()

def grabTree(filename):

import pickle

fr = open(filename)

return pickle.load(fr)

if \_\_name\_\_ == '\_\_main\_\_':

fr = open('play.tennies.txt')

lenses =[inst.strip().split(' ') for inst in fr.readlines()]

lensesLabels = ['outlook','temperature','huminidy','windy']

lensesTree =createTree(lenses,lensesLabels)

treePlotter.createPlot(lensesTree)

#!/usr/bin/python

#encoding:utf-8

#treePlotter.py

import matplotlib.pyplot as plt

decisionNode = dict(boxstyle="sawtooth", fc="0.8") #定义文本框与箭头的格式

leafNode = dict(boxstyle="round4", fc="0.8")

arrow\_args = dict(arrowstyle="<-")

def getNumLeafs(myTree): #获取树叶节点的数目

numLeafs = 0

firstStr = myTree.keys()[0]

secondDict = myTree[firstStr]

for key in secondDict.keys():

if type(secondDict[key]).\_\_name\_\_=='dict':#测试节点的数据类型是不是字典，如果是则就需要递归的调用getNumLeafs()函数

numLeafs += getNumLeafs(secondDict[key])

else: numLeafs +=1

return numLeafs

def getTreeDepth(myTree): #获取树的深度

maxDepth = 0

firstStr = myTree.keys()[0]

secondDict = myTree[firstStr]

for key in secondDict.keys():

if type(secondDict[key]).\_\_name\_\_=='dict':#test to see if the nodes are dictonaires, if not they are leaf nodes

thisDepth = 1 + getTreeDepth(secondDict[key])

else: thisDepth = 1

if thisDepth > maxDepth: maxDepth = thisDepth

return maxDepth

# 绘制带箭头的注释

def plotNode(nodeTxt, centerPt, parentPt, nodeType):

createPlot.ax1.annotate(nodeTxt, xy=parentPt, xycoords='axes fraction',

xytext=centerPt, textcoords='axes fraction',

va="center", ha="center", bbox=nodeType, arrowprops=arrow\_args )

#计算父节点和子节点的中间位置，在父节点间填充文本的信息

def plotMidText(cntrPt, parentPt, txtString)