

**Department of Computer Science and Engineering**

**(UG Studies)**

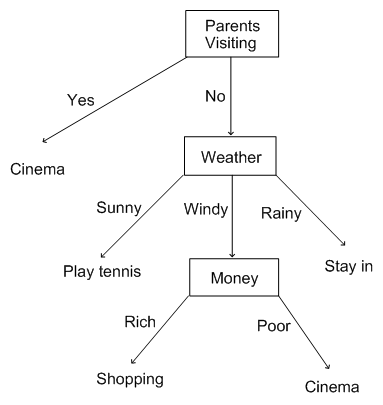
**PES University, Bangalore-560085**

|  |  |
| --- | --- |
| **Session :** Aug - Dec 2017  **Credits :** 0-0-2-0-1 | UE14CS405 : Machine Learning Lab |
| **Lab # :** 04 | Construct/Visualize a Decision Tree using Information Gain (Entropy) attributes on a (at least) 32- size data set with (at least) 8- Attributes |

**Theory:**

Imagine you only ever do four things at the weekend: go shopping, watch a movie, play tennis or just stay in. What you do depends on three things: the weather (windy, rainy or sunny); how much money you have (rich or poor) and whether your parents are visiting. You say to your yourself: if my parents are visiting, we'll go to the cinema. If they're not visiting and it's sunny, then I'll play tennis, but if it's windy, and I'm rich, then I'll go shopping. If they're not visiting, it's windy and I'm poor, then I will go to the cinema. If they're not visiting and it's rainy, then I'll stay in.

To remember all this, you draw a flowchart which will enable you to read off your decision. We call such diagrams decision trees. A suitable decision tree for the weekend decision choices would be as follows:



We can see why such diagrams are called trees, because, while they are admittedly upside down, they start from a root and have branches leading to leaves (the tips of the graph at the bottom). Note that the leaves are always decisions, and a particular decision might be at the end of multiple branches (for example, we could choose to go to the cinema for two different reasons).

**Data Set Information**

Predicting the age of abalone from physical measurements. The age of abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope -- a boring and time-consuming task. Other measurements, which are easier to obtain, are used to predict the age.

Attribute Information:

Given is the attribute name, attribute type, the measurement unit and a brief description. The number of rings is the value to predict: either as a continuous value or as a classification problem.   
  
Name / Data Type / Measurement Unit / Description   
-----------------------------   
Sex / nominal / -- / M, F, and I (infant)   
Length / continuous / mm / Longest shell measurement   
Diameter / continuous / mm / perpendicular to length   
Height / continuous / mm / with meat in shell   
Whole weight / continuous / grams / whole abalone   
Shucked weight / continuous / grams / weight of meat   
Viscera weight / continuous / grams / gut weight (after bleeding)   
Shell weight / continuous / grams / after being dried   
Rings / integer / gives the age in years

**Code:**

from math import log

import operator

import matplotlib

matplotlib.use('Agg')

import matplotlib.pyplot as plt

def createDataSet():

dataSet = [['M',0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15],

['M',0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7],

['F',0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9],

['M',0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10],

['I',0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7],

['I',0.425,0.3,0.095,0.3515,0.141,0.0775,0.12,8],

['F',0.53,0.415,0.15,0.7775,0.237,0.1415,0.33,20],

['F',0.545,0.425,0.125,0.768,0.294,0.1495,0.26,16],

['M',0.475,0.37,0.125,0.5095,0.2165,0.1125,0.165,9],

['F',0.55,0.44,0.15,0.8945,0.3145,0.151,0.32,19],

['F',0.525,0.38,0.14,0.6065,0.194,0.1475,0.21,14]]

labels = ['Sex','Length','Diameter','Height','Whole weight','Shucked weight','Viscera weight','Shell weight','Rings']

# change to discrete values

return dataSet, labels

def calcShannonEnt(dataSet):

numEntries = len(dataSet)

labelCounts = {}

for featVec in dataSet: # the the number of unique elements and their occurance

currentLabel = featVec[-1]

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

labelCounts[currentLabel] += 1

shannonEnt = 0.0

for key in labelCounts:

prob = float(labelCounts[key]) / numEntries

shannonEnt -= prob \* log(prob, 2) # log base 2

return shannonEnt

def splitDataSet(dataSet, axis, value):

retDataSet = []

for featVec in dataSet:

if featVec[axis] == value:

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

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

retDataSet.append(reducedFeatVec)

return retDataSet

def chooseBestFeatureToSplit(dataSet):

numFeatures = len(dataSet[0]) - 1 # the last column is used for the labels

baseEntropy = calcShannonEnt(dataSet)

bestInfoGain = 0.0;

bestFeature = -1

for i in range(numFeatures): # iterate over all the features

featList = [example[i] for example in dataSet] # create a list of all the examples of this feature

uniqueVals = set(featList) # get a set of unique values

newEntropy = 0.0

for value in uniqueVals:

subDataSet = splitDataSet(dataSet, i, value)

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

newEntropy += prob \* calcShannonEnt(subDataSet)

infoGain = baseEntropy - newEntropy # calculate the info gain; ie reduction in entropy

"""

print("feature : " + str(i))

print("baseEntropy : "+str(baseEntropy))

print("newEntropy : " + str(newEntropy))

print("infoGain : " + str(infoGain))

"""

if (infoGain > bestInfoGain): # compare this to the best gain so far

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

bestFeature = i

return bestFeature # returns an integer

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]

def createTree(dataSet, labels):

# extracting data

classList = [example[-1] for example in dataSet]

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

return classList[0] # stop splitting when all of the classes are equal

if len(dataSet[0]) == 1: # stop splitting when there are no more features in dataSet

return majorityCnt(classList)

# use Information Gain

bestFeat = chooseBestFeatureToSplit(dataSet)

bestFeatLabel = labels[bestFeat]

#build a tree recursively

myTree = {bestFeatLabel: {}}

#print("myTree : "+labels[bestFeat])

del (labels[bestFeat])

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

#print("featValues: "+str(featValues))

uniqueVals = set(featValues)

#print("uniqueVals: " + str(uniqueVals))

for value in uniqueVals:

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

#print("subLabels"+str(subLabels))

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

#print("myTree : " + str(myTree))

return myTree

def classify(inputTree, featLabels, testVec):

firstStr = inputTree.keys()[0]

#print("fistStr : "+firstStr)

secondDict = inputTree[firstStr]

#print("secondDict : " + str(secondDict))

featIndex = featLabels.index(firstStr)

#print("featIndex : " + str(featIndex))

key = testVec[featIndex]

#print("key : " + str(key))

valueOfFeat = secondDict[key]

#print("valueOfFeat : " + str(valueOfFeat))

if isinstance(valueOfFeat, dict):

#print("is instance: "+str(valueOfFeat))

classLabel = classify(valueOfFeat, featLabels, testVec)

else:

#print("is Not instance: " + valueOfFeat)

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)

decisionNode = dict(boxstyle="sawtooth", fc="0.8")

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

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

def getNumLeafs(myTree):

numLeafs = 0

firstStr = list(myTree)[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

numLeafs += getNumLeafs(secondDict[key])

else: numLeafs +=1

return numLeafs

def getTreeDepth(myTree):

maxDepth = 0

firstStr = list(myTree)[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

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 )

def plotMidText(cntrPt, parentPt, txtString):

xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]

yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]

createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="center", rotation=30)

def plotTree(myTree, parentPt, nodeTxt):#if the first key tells you what feat was split on

numLeafs = getNumLeafs(myTree) #this determines the x width of this tree

depth = getTreeDepth(myTree)

firstStr = list(myTree)[0] #the text label for this node should be this

cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)

plotMidText(cntrPt, parentPt, nodeTxt)

plotNode(firstStr, cntrPt, parentPt, decisionNode)

secondDict = myTree[firstStr]

plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD

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

plotTree(secondDict[key],cntrPt,str(key)) #recursion

else: #it's a leaf node print the leaf node

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

#if you do get a dictonary you know it's a tree, and the first element will be another dict

def createPlot(inTree):

fig = plt.figure(1, facecolor='blue')

fig.clf()

axprops = dict(xticks=[], yticks=[])

createPlot.ax1 = plt.subplot(111, frameon=False, \*\*axprops) #no ticks

#createPlot.ax1 = plt.subplot(111, frameon=False) #ticks for demo puropses

plotTree.totalW = float(getNumLeafs(inTree))

plotTree.totalD = float(getTreeDepth(inTree))

plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;

plotTree(inTree, (0.5,1.0), '')

plt.show()

plt.savefig("temp.png")

print("Finished")

# collect data

myDat, labels = createDataSet()

mytree = createTree(myDat, labels)

print(mytree)

#visualize decision tree

createPlot(mytree)

**TO DO:**

**1)** Understand the given code and calculate entropy manually for atleast two attributes.

2) Modify the given code using gain ratio.

Note: Gain ratio: a modification of the information gain that reduces its bias

**Learning Outcomes:**

Decision tree builds a model called tree using training data, once the model is ready new test tuple with out the class label can be passed to the model and trace the path to get the lable for the test data.