Bài thi môn: Trí tuệ nhân tạo

Họ và tên: Lê Thanh Tuấn

Mã số sinh viên: 61134623

Lớp : 60 CNTT1

Bài làm

Bài 2

Thuộc tính Giới tính

VGiới\_tính(Nam) = (D(Nam, Mua), D(Nam, Ko mua)) = (1/2, 1/2)

VGiới\_tính(Nữ) = (D(Nữ, Mua), D(Nữ, Ko mua)) = (1/2, 1/2)

Thuộc tính Quốc tịch

VQuốc\_tịch(Việt Nam) = (D(Việt Nam, Mua), D(Việt Nam, Ko mua)) = (2/3, 1/3)

VQuốc\_tịch(Hàn Quốc) = (D(Hàn Quốc, Mua), D(Hàn Quốc, Ko mua)) =(0,1)=> vector đơn vị

VQuốc\_tịch(Mỹ) = (D(Mỹ, Mua), D(Mỹ, Ko mua)) = (2/3, 1/3)

Thuộc tính Màu sắc

VMàu\_sắc(Đỏ) = (D(Đỏ, Mua), D(Đỏ, Ko mua)) = (1/2, 1/2)

VMàu\_sắc(Hồng) = (D(Hồng, Mua), D(Hồng, Ko mua)) = (1/2, 1/2)

VMàu\_sắc(Tím) = (D(Tím, Mua), D(Tím, Ko mua)) = (1/2, 1/2)

Thuộc tính Size

VSize(Nhỏ) = (D(Nhỏ, Mua), D(Nhỏ, Ko mua)) = (1/2, 1/2)

VSize(Vừa) = (D(Vừa, Mua), D(Vừa, Ko mua)) = (1/2, 1/2)

VSize(Lớn) = (D(Lớn, Mua), D(Lớn, Ko mua)) = (1/2, 1/2)

* Như vậy thuộc tính Quốc tịch có nhiều vector đơn vị nhất nên chọn làm thuộc tính phân nhánh

Quốc tịch Việt Nam

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| STT | Giới tính | Màu sắc | Size | Quyết định |
| 1 | Nam | Đỏ | Nhỏ | Mua |
| 5 | Nam | Đỏ | Nhỏ | Ko mua |
| 6 | Nam | Hồng | Vừa | Mua |

Thuộc tính Giới Tính

VGiới\_tính(Nam) = (D(Nam, Mua), D(Nam, Ko mua)) = (2/3,1/3)

Thuộc tính Màu sắc

VMàu\_sắc(Đỏ) = (D(Đỏ, Mua), D(Đỏ, Ko mua)) = (1/2, 1/2)

VMàu\_sắc(Hồng) = (D(Hồng, Mua), D(Hồng, Ko mua)) = (1,0)=> vector đơn vị

Thuộc tính Size

VSize(Nhỏ) = (D(Nhỏ, Mua), D(Nhỏ, Ko mua)) = (1/2, 1/2)

VSize(Vừa) = (D(Vừa, Mua), D(Vừa, Ko mua)) = (1,0)=> vector đơn vị

Quốc tịch Mỹ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| STT | Giới tính | Màu sắc | Size | Quyết định |
| 3 | Nữ | Tím | Lớn | Ko mua |
| 4 | Nữ | Tím | Vừa | Mua |
| 8 | Nữ | Đỏ | Lớn | Mua |

Thuộc tính Giới tính

VGiới\_tính(Nữ) = (D(Nữ, Mua), D(Nữ, Ko mua)) = (2/3, 1/3)

Thuộc tính Màu sắc

VMàu\_sắc(Tím) = (D(Tím, Mua), D(Tím, Ko mua)) = (1/2, 1/2)

VMàu\_sắc(Đỏ) = (D(Đỏ, Mua), D(Đỏ, Ko mua)) = (1, 0)

Thuộc tính Size

VSize(Vừa) = (D(Vừa, Mua), D(Vừa, Ko mua)) = (2/3, 1/3)

VSize(Vừa) = (D(Vừa, Mua), D(Vừa, Ko mua)) = (2/3, 1/3)

* Màu sắc có nhiều vector đơn vị nhất nên chọn thuộc tính Màu sắc làm thuộc tính phân hoạch.

Việt Nam Nam

Mỹ

Hàn

Không mua

Hồngg

Đỏ

Mua

Mua

if( Quốc tịch is Hàn Quốc) then (ko mua)

if( Quốc tịch isViệt Nam) and (Màu sắc is hồng ) then ( mua)

if( Quốc tịch is Mỹ) and () AND (Màu sắc is Đỏ ) then (mua)

2/b

**import math**

**class C45:**

"""Creates a decision tree with C4.5 algorithm"""

**def \_\_init\_\_(self, pathToData,pathToNames):**

**self.filePathToData = pathToData**

**self.filePathToNames = pathToNames**

**self.data = []**

**self.classes = []**

**self.numAttributes = -1**

**self.attrValues = {}**

**self.attributes = []**

**self.tree = None**

def fetchData(self):

with open(self.filePathToNames, "r") as file:

classes = file.readline()

self.classes = [x.strip() for x in classes.split(",")]

#add attributes

for line in file:

[attribute, values] = [x.strip() for x in line.split(":")]

values = [x.strip() for x in values.split(",")]

self.attrValues[attribute] = values

self.numAttributes = len(self.attrValues.keys())

self.attributes = list(self.attrValues.keys())

with open(self.filePathToData, "r") as file:

for line in file:

row = [x.strip() for x in line.split(",")]

if row != [] or row != [""]:

self.data.append(row)

def preprocessData(self):

for index,row in enumerate(self.data):

for attr\_index in range(self.numAttributes):

if(not self.isAttrDiscrete(self.attributes[attr\_index])):

self.data[index][attr\_index] = float(self.data[index][attr\_index])

def printTree(self):

self.printNode(self.tree)

def printNode(self, node, indent=""):

if not node.isLeaf:

if node.threshold is None:

#discrete

for index,child in enumerate(node.children):

if child.isLeaf:

print(indent + node.label + " = " + attributes[index] + " : " + child.label)

else:

print(indent + node.label + " = " + attributes[index] + " : ")

self.printNode(child, indent + " ")

else:

#numerical

leftChild = node.children[0]

rightChild = node.children[1]

if leftChild.isLeaf:

print(indent + node.label + " <= " + str(node.threshold) + " : " + leftChild.label)

else:

print(indent + node.label + " <= " + str(node.threshold)+" : ")

self.printNode(leftChild, indent + " ")

if rightChild.isLeaf:

print(indent + node.label + " > " + str(node.threshold) + " : " + rightChild.label)

else:

print(indent + node.label + " > " + str(node.threshold) + " : ")

self.printNode(rightChild , indent + " ")

**def generateTree(self):**

**self.tree = self.recursiveGenerateTree(self.data, self.attributes)**

**def recursiveGenerateTree(self, curData, curAttributes):**

**allSame = self.allSameClass(curData)**

**if len(curData) == 0:**

**#Fail**

**return Node(True, "Fail", None)**

**elif allSame is not False:**

**#return a node with that class**

**return Node(True, allSame, None)**

**elif len(curAttributes) == 0:**

**#return a node with the majority class**

**majClass = self.getMajClass(curData)**

**return Node(True, majClass, None)**

**else:**

**(best,best\_threshold,splitted) = self.splitAttribute(curData, curAttributes)**

**remainingAttributes = curAttributes[:]**

**remainingAttributes.remove(best)**

**node = Node(False, best, best\_threshold)**

**node.children = [self.recursiveGenerateTree(subset, remainingAttributes) for subset in splitted]**

**return node**

**def getMajClass(self, curData):**

**freq = [0]\*len(self.classes)**

**for row in curData:**

**index = self.classes.index(row[-1])**

**freq[index] += 1**

**maxInd = freq.index(max(freq))**

**return self.classes[maxInd]**

**def allSameClass(self, data):**

**for row in data:**

**if row[-1] != data[0][-1]:**

**return False**

**return data[0][-1]**

**def isAttrDiscrete(self, attribute):**

**if attribute not in self.attributes:**

**raise ValueError("Attribute not listed")**

**elif len(self.attrValues[attribute]) == 1 and self.attrValues[attribute][0] == "continuous":**

**return False**

**else:**

**return True**

**def splitAttribute(self, curData, curAttributes):**

**splitted = []**

**maxEnt = -1\*float("inf")**

**best\_attribute = -1**

**#None for discrete attributes, threshold value for continuous attributes**

**best\_threshold = None**

**for attribute in curAttributes:**

**indexOfAttribute = self.attributes.index(attribute)**

**if self.isAttrDiscrete(attribute):**

**#split curData into n-subsets, where n is the number of**

**#different values of attribute i. Choose the attribute with**

**#the max gain**

**valuesForAttribute = self.attrValues[attribute]**

**subsets = [[] for a in valuesForAttribute]**

**for row in curData:**

**for index in range(len(valuesForAttribute)):**

**if row[i] == valuesForAttribute[index]:**

**subsets[index].append(row)**

**break**

**e = gain(curData, subsets)**

**if e > maxEnt:**

**maxEnt = e**

**splitted = subsets**

**best\_attribute = attribute**

**best\_threshold = None**

**else:**

**#sort the data according to the column.Then try all**

**#possible adjacent pairs. Choose the one that**

**#yields maximum gain**

**curData.sort(key = lambda x: x[indexOfAttribute])**

**for j in range(0, len(curData) - 1):**

**if curData[j][indexOfAttribute] != curData[j+1][indexOfAttribute]:**

**threshold = (curData[j][indexOfAttribute] + curData[j+1][indexOfAttribute]) / 2**

**less = []**

**greater = []**

**for row in curData:**

**if(row[indexOfAttribute] > threshold):**

**greater.append(row)**

**else:**

**less.append(row)**

**e = self.gain(curData, [less, greater])**

**if e >= maxEnt:**

**splitted = [less, greater]**

**maxEnt = e**

**best\_attribute = attribute**

**best\_threshold = threshold**

**return (best\_attribute,best\_threshold,splitted)**

**def gain(self,unionSet, subsets):**

**#input : data and disjoint subsets of it**

**#output : information gain**

**S = len(unionSet)**

**#calculate impurity before split**

**impurityBeforeSplit = self.entropy(unionSet)**

**#calculate impurity after split**

**weights = [len(subset)/S for subset in subsets]**

**impurityAfterSplit = 0**

**for i in range(len(subsets)):**

**impurityAfterSplit += weights[i]\*self.entropy(subsets[i])**

**#calculate total gain**

**totalGain = impurityBeforeSplit - impurityAfterSplit**

**return totalGain**

**def entropy(self, dataSet):**

**S = len(dataSet)**

**if S == 0:**

**return 0**

**num\_classes = [0 for i in self.classes]**

**for row in dataSet:**

**classIndex = list(self.classes).index(row[-1])**

**num\_classes[classIndex] += 1**

**num\_classes = [x/S for x in num\_classes]**

**ent = 0**

**for num in num\_classes:**

**ent += num\*self.log(num)**

**return ent\*-1**

def log(self, x):

if x == 0:

return 0

else:

return math.log(x,2)

**class Node:**

**def \_\_init\_\_(self,isLeaf, label, threshold):**

**self.label = label**

**self.threshold = threshold**

**self.isLeaf = isLeaf**

**self.children = []**

Bài 3:

robot\_ear = speech\_recognition.Recognizer()

robot\_mouth = pyttsx3.init()

robot\_brain = ""

while True: # cái này để mình và robot giao tiếp liên tục thay vì nói 1 câu chương trình đã kết thúc.

with speech\_recognition.Microphone() as mic:

print("Robot: I'm Listening")

audio = robot\_ear.listen(mic)

print("Robot:...")

try:

you = robot\_ear.recognizer\_google(audio)

except:

you = ""

if you == "":

robot\_brain = "I can't hear you, try again"

elif "Hello" in you:

robot\_brain: "Hello nice to meet you"

elif "goodbye" in you: ## đoạn này khi nói goodbye thì chương trình sẽ tắt thay vì mở liên tục khi ở phía trên

robot\_brain = "Good Bye"

break

else:

robot\_brain = "I'm fine thank you and you"

print("Robot:" + robot\_brain)

robot\_mouth.say(robot\_brain)

robot\_mouth.runAndWait()