Laboratory 3 – Binary Trees

CS 2302 – Data structures summer 2019

Dilan Ramirez

INSTRUCTOR: OLAC FUENTES

TEACHING ASSISTANTS (TA): ISMAEL VILLANUEVA-MIRANDA

JULY 9, 2019

Spring 2019

Contents

[Introduction 2](#_Toc14115501)

[Implementation 2](#_Toc14115502)

[Running Time 4](#_Toc14115503)

[Experimental results 4](#_Toc14115504)

[Conclusion 5](#_Toc14115505)

[Appendix 6](#_Toc14115506)

[Honesty Certification 9](#_Toc14115507)

# Introduction

The file magic04.txt contains data from a gamma ray detection experiment. Each line in the file represents an observation, with the first ten items, all floating point numbers, describing the data collected by the detector, and the last item, a character (g or h), indicating whether the detection corresponds to a gamma ray (g) or not (h). The task for this lab is to construct a decision tree, as described in class, for this problem. The program decisiontree.py provides code that reads the data, splits it into training and testing parts, and implements a one-node decision tree from the training data to classify the test data. For this lab, the decision tree functions will be extended to build trees with more than one node and use these trees to classify the data, hopefully improving accuracy.

1. Extend the BuildDT function. In the current implementation, leftChild and rightChild are classification labels (0 or 1), in the extended implementation, they should be references to decision trees, provided the dataset is large enough and the goal accuracy has not been attained.
2. Experiment with different values of parameters to obtain the highest possible accuracy on the test set.
3. Display the following statistics about the tree generated: number of nodes, number of times each attribute is used for splitting, and tree height.
4. Generate multiple trees from the training data and average the results to make predictions. Determine if accuracy can be improved doing this

# Implementation

The first task of the lab was to extend the BuildDT function to do build left decision subtree recursively if left\_acc is less than goal accuracy and left\_size is greater than min\_size. After that, do the same for the right side. To do that I use these conditions to create the left and right sides of the tree using the information of test\_data. After following the instructions, the first recursive call was to the left side,

if left\_acc < goal\_acc and left\_size > min\_size:

left\_child = buildDT(attributes[left\_best\_split], target[left\_best\_split], n\_splits, goal\_acc, min\_size)

The next recursive call was to the right,

if right\_acc < goal\_acc and right\_size > min\_size:

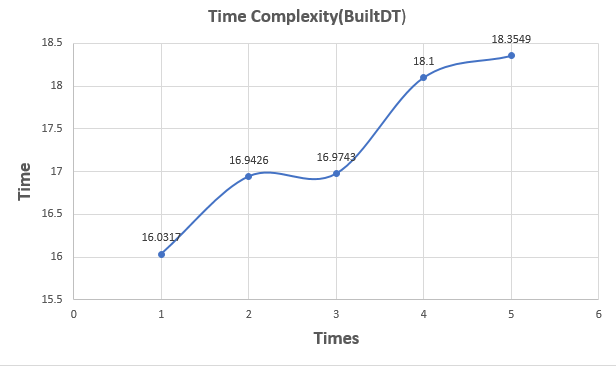
right\_child = buildDT(attributes[right\_best\_split], target[right\_best\_split], n\_splits, goal\_acc, min\_size)

if the left or right accuracies are equal to the goal accuracies the next call will be the only one to be executed,

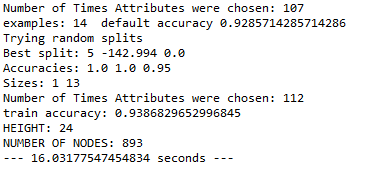
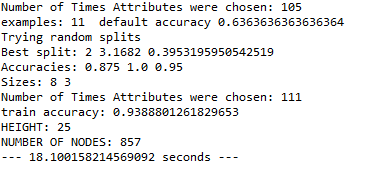
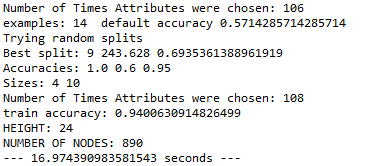
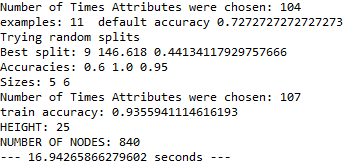
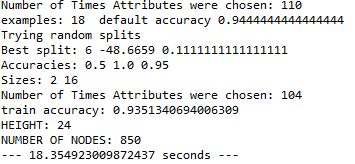
return decisionTreeNode(best\_a, best\_thr, left\_child, right\_child)

For the “Height” function I use the same getHeight() function that we were used before, but now I changed the base case of the function because the simplest case is if the tree is either 0 or 1. After that, I call the function recursively for the left and right side plus 1 and get the max for these two sides. For the function number of nodes I used the same function that I have been using before, but in this case the case base was different if the tree is either 0 or 1. After that, the function adds the quantity of nodes in the left side plus the right side plus 1 that is the root of the tree. For the number of times each attribute is used for splitting what I did was to add a counter in the buildDT() function. These counter keeps track of the times each attribute is chosen and used by the function.

# Running Time



# Experimental results

1. Experiment with different values of parameters to obtain the highest possible accuracy on the test set
2. Display the following statistics about the tree generated: number of nodes, number of times each attribute is used for splitting, and tree height
   1. 
   2. 
   3. 
   4. 
   5. 

# Conclusion

By doing this lab I learned how to use a binary search tree for another thing. In this case, we use a binary search tree to store values from observation. I leaned how to use a binary search tree with other parameters and not only left, right and item. In this case, we use attribute, left and right. I learned as well how to manage a function when the tree is not None.in this case, the base cases were if decisionTreeNode == 0 or decisionTreeNode == 1.

# Appendix

import numpy as np

import random

import time

start\_time = time.time()

class decisionTreeNode(object):

# Constructor

def \_\_init\_\_(self, att, thr, left, right):

self.attribute = att

self.threshold = thr

# left and right are either bynary classifications, or references to

# decision tree nodes

self.left = left

self.right = right

def entropy(l,m=[]):

ent = 0

for p in [l,m]:

if len(p)>0:

pp = sum(p)/len(p)

pn = 1 -pp

if pp<1 and pp>0:

ent -= len(p)\*(pp\*np.log2(pp)+pn\*np.log2(pn))

ent = ent/(len(l)+len(m))

return ent

def classify(DT, atts):

if atts[DT.attribute] < DT.threshold:

if DT.left in [0,1]:

return DT.left

else:

return classify(DT.left, atts)

else:

if DT.right in [0,1]:

return DT.right

else:

return classify(DT.right, atts)

def buildDT(attributes, target, n\_splits, goal\_acc, min\_size):

# Builds a one-node decision tree to classify data

print('examples:',len(target), ' default accuracy',max([np.mean(target),1-np.mean(target)]))

print('Trying random splits')

best\_ent = 1

min\_att\_val = np.min(attributes,axis=0)

count = 0

for i in range(n\_splits):

while True:

a = random.randrange(attributes.shape[1])

ex = random.randrange(attributes.shape[0])

count += 1

thr = attributes[ex,a]

if thr>min\_att\_val[a]: # making sure we don't have an empty splits

break

less = attributes[:,a] < thr

more = ~ less

tgt\_less = target[less]

tgt\_more = target[more]

if len(less) == 0 or len(more) == 0:

ent = 1

else:

ent = entropy(tgt\_less,tgt\_more)

# print(i,a,thr,ent) # Used for debugging

if ent < best\_ent:

best\_ent, best\_a, best\_thr = ent, a, thr

left\_child = int(np.mean(tgt\_less)+.5)

right\_child = int(np.mean(tgt\_more)+.5)

left\_best\_split = less

right\_best\_split = more

# print(a,thr,ent) # Used for debugging

print('Best split:',best\_a,best\_thr,best\_ent)

ml = np.mean(target[left\_best\_split])

left\_acc = max([ml,1-ml])

left\_size = np.sum(left\_best\_split)

mm = np.mean(target[right\_best\_split])

right\_acc = max([mm,1-mm])

right\_size = np.sum(right\_best\_split)

print('Accuracies:',left\_acc,right\_acc, goal\_acc)

print('Sizes:',left\_size,right\_size)

print('Number of Times Attributes were chosen:',count)

# Modify as follows:

# if left\_acc is less than goal accuracy and left\_size is greater than min\_size

# build left decision subtree recursively

# do the same for the right side

if left\_acc < goal\_acc and left\_size > min\_size:

left\_child = buildDT(attributes[left\_best\_split], target[left\_best\_split], n\_splits, goal\_acc, min\_size)

if right\_acc < goal\_acc and right\_size > min\_size:

right\_child = buildDT(attributes[right\_best\_split], target[right\_best\_split], n\_splits, goal\_acc, min\_size)

return decisionTreeNode(best\_a, best\_thr, left\_child, right\_child)

def Height(decisionTreeNode):

if decisionTreeNode == 0 or decisionTreeNode == 1:

return 0

return max(Height(decisionTreeNode.left), Height(decisionTreeNode.right)) + 1

def NumNodes(decisionTreeNode):

if decisionTreeNode == 0 or decisionTreeNode == 1:

return 0

return NumNodes(decisionTreeNode.left) + 1 + NumNodes(decisionTreeNode.right)

attributes = []

target = []

infile = open("magic04.txt","r")

for line in infile:

target.append(int(line[-2:-1] =='g'))

attributes.append(np.fromstring(line[:-2], dtype=float,sep=','))

infile.close()

attributes = np.array(attributes)

target = np.array(target)

#Split data into training and testing

ind = np.random.permutation(len(target))

split\_ind = int(len(target)\*0.8)

train\_data = attributes[ind[:split\_ind]]

test\_data = attributes[ind[split\_ind:]]

train\_target = target[ind[:split\_ind]]

test\_target = target[ind[split\_ind:]]

dt = buildDT(train\_data, train\_target, 100, 0.95, 10)

L = []

train\_pred = np.zeros(train\_target.shape, dtype=int)

for i in range(len(train\_pred)):

train\_pred[i] = classify(dt, train\_data[i])

train\_acc = np.sum(train\_pred==train\_target)/len(train\_pred)

print('train accuracy:', train\_acc)

test\_pred = np.zeros(test\_target.shape, dtype=int)

for i in range(len(test\_pred)):

test\_pred[i] = classify(dt, test\_data[i])

test\_acc = np.sum(test\_pred == test\_target)/len(test\_pred)

L.append(train\_acc)

print(Accuracy(dt))

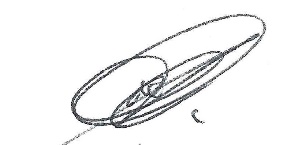
print("HEIGHT:", Height(dt))

print("NUMBER OF NODES:", NumNodes(dt))

print("--- %s seconds ---" % (time.time() - start\_time))

# Honesty Certification

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provide inappropriate assistance to any student in the class.

 07/ 11 / 2019

Dilan Ramirez Date