Nathan Aun

7/9/19

Lab #3 – Decision Trees

CS2302 – Data Structures

Summer 2019

Introduction

The idea posed for this lab was to make a decision tree that will be able to predict whether or not a wave is a gamma ray based on 10 attributes. For our purposes, what these attributes represent is unimportant and we are instead just going to try to have the highest accuracy on the test set as possible. We need to finish the buildDT method and make our tree as accurate as possible, while also implementing methods that will tell us how many times we’ve split by a certain attribute, the number of nodes in our final tree, and lastly the height of our newly made decision tree.

Proposed Solution

The first part of this lab is to make all the methods and really we do not need to do much in the buildDT method in order to get it functioning. Adding a recursive call to build the tree to the left and right is really all that needs to be done in order to get this working. It will only build a subtree if the accuracy is not at the target and there are enough elements to keep building an accurate tree. This second parameter is important because, while we can get really high predictions on our training set, it will only be that high on our training data and it won’t transfer to actual real world data which is what we need it to.

The other methods don’t have much to implement either. Their all recursive methods with the base case of DT == 0 or DT ==1 because that is always going to be the value of a “leaf” node in this implementation of a decision tree. Num nodes will add 1 at the base case, and every recursive call also add 1 to the total. Height does the same thing except it does a max between the two subtrees as it builds, so we only get the height of the tree and not every node. Attribute split by has an array of size 10 (number of attributes) and it will add 1 to the index of the attribute used to split. This is useful because it will let us know how many times each attribute was used without needing to use multiple variables.

Experimental Results

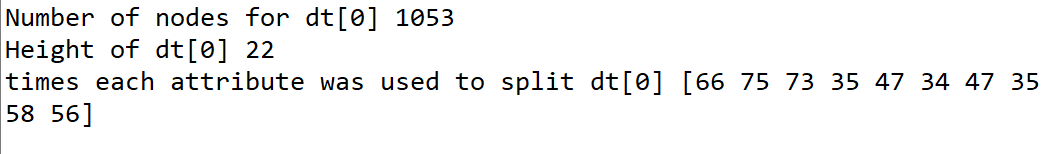
After the methods were implemented it was time to determine which attributes affected the accuracy and to manipulate them to obtain the highest possible accuracy. The general trend I was able to see was that the lower number minimum elements was the higher the train accuracy was, but this had little bearing over the test accuracy, which was usually around 82% on these tests. Having this number too high also led to some very inaccurate results, so it couldn’t be too big. The number of splits also had the next highest bearing on not only accuracy but also runtime. The more splits that happen the slower the program goes, so it can take a very long time to build trees that have a lot of splits. The number of splits alone didn’t seem to hugely affect the accuracy but it did generally seem to change the accuracy a bit. The last thing to change was the target accuracy and this also did not affect the tree very much. If it was set below 85% then it did have an impact but since it didn’t get to 85% most of the time any value higher didn’t have much of a bearing on it. Here is the general results in table form:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Num\_Splits | Target\_accuracy | MinElements | Train accuracy | Test accuracy |
| 1000 | 0.95 | 100 | 0.94 | 0.83 |
| 1000 | 0.95 | 150 | 0.96 | 0.829 |
| 1000 | 0.95 | 200 | 0.85 | 0.849 |
| 1000 | 0.95 | 250 | 0.85 | 0.847 |
| 1000 | 0.95 | 300 | 0.85 | 0.84 |
|  |  |  |  |  |
| 100 | 0.95 | 5 | 0.96 | 0.823 |
| 100 | 0.95 | 10 | 0.935 | 0.83333333 |
| 100 | 0.95 | 15 | 0.92 | 0.838 |
| 100 | 0.95 | 20 | 0.91 | 0.839 |
| 100 | 0.95 | 25 | 0.9 | 0.84 |
| 100 | 0.95 | 30 | 0.89 | 0.84 |
|  |  |  |  |  |
| 10000 | 0.95 | 500 | 0.84 | 0.816 |
| 10000 | 0.95 | 1000 | 0.838 | 0.825 |
| 10000 | 0.95 | 1500 | 0.83 | 0.8 |
| 10000 | 0.95 | 2000 | 0.82 | 0.81 |
| 10000 | 0.95 | 2500 | 0.81 | 0.8 |
| 10000 | 0.95 | 3000 | 0.799 | 0.774 |
|  |  |  |  |  |
| 5000 | 0.95 | 500 | 0.84 | 0.823 |
| 5000 | 0.95 | 750 | 0.84 | 0.833 |
| 5000 | 0.95 | 1000 | 0.83 | 0.826 |
| 5000 | 0.95 | 1250 | 0.829 | 0.824 |
| 5000 | 0.95 | 1500 | 0.821 | 0.824 |
| 5000 | 0.95 | 1750 | 0.821 | 0.811 |
|  |  |  |  |  |
|  |  |  |  |  |
| 1000 | 0.95 | 200 | 0.85 | 0.849 |
| 1000 | 0.9 | 200 | 0.85 | 0.841 |
| 1000 | 0.85 | 200 | 0.844 | 0.842 |
| 1000 | 0.8 | 200 | 0.83 | 0.817 |

After that it was time to see how averaging multiple trees together could work for the accuracy. The highest any individual tree was able to get was approximately 85% on the test data, so if any of these averages are more than that then this is a success. I used 51 trees for most tests and the last one I chose to use 501 trees to see if it would make a dramatic difference. The best accuracy I was able to obtain when averaging multiple trees together was around 87.6% which admittedly was not as good as I hoped, but it still was an increase so it proves that averaging multiple trees together does help get a higher accuracy. Interestingly enough I used the same parameters that reached 87.6% with 51 trees and with 501 trees and it actually became less accurate for 501 trees, with a test accuracy of 87.1%. This is likely a result of random chance more than anything, since each tree is build randomly each time, but it is interesting that more trees did not equal higher accuracy.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Num\_Splits | Target\_accuracy | MinElements | Train accuracy | Test accuracy | Number of trees |
| 100 | 0.95 | 25 | 0.933 | 0.876 | 51 trees |
| 100 | 0.95 | 20 | 0.944 | 0.873 | 51 trees |
| 100 | 0.95 | 30 | 0.924 | 0.865 | 51 trees |
| 1000 | 0.95 | 200 | 0.863 | 0.847 | 51 trees |
| 100 | 0.95 | 25 | 0.936 | 0.871 | 501 trees |

Lastly, here’s proof that my extra methods work:



Conclusions

Overall this has been a very interesting lab and it’s really interesting to see something like machine learning in this curriculum. Given more time, I’d probably run more tests than I did to see more clearly how each attribute affects the accuracy but overall I feel like I generally understand how decision trees work and how they can be used in machine learning.

Appendix

Source code is as follows:

import numpy as np

import random

'''

Course: CS2302 Data Structures

Author: Nathan Aun

Assignment: Lab 3 - Decision Trees

Instructor: Dr. Fuentes

TA: Ismael Villanueva-Miranda

Date of Last Modification: 7/8/2019

Purpose of Program: To use decision trees to predict if a given

wave is a gamma ray based on 1/10 attributes, chosen at random

'''

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)

#Num nodes will take a decision tree node and return the number of nodes in that tree

def NumNodes(DT):

if DT == 0 or DT == 1:

return 1

temp = 0

temp += NumNodes(DT.left) + 1 + NumNodes(DT.right)

return temp

#Height will take a decision tree node and return the height of that tree

def Height(DT):

if DT == 0 or DT == 1:

return 1

height = max(Height(DT.left) + 1,Height(DT.right) + 1)

return height

#AttributeSplitBy will take a Decision Tree and an array of size 10, and will increment

#The index of the value used to split in that place

def AttributeSplitBy(DT, arr):

if DT == 0 or DT == 1:

return arr

arr[DT.attribute] += 1

arr = AttributeSplitBy(DT.left, arr)

arr = AttributeSplitBy(DT.right, arr)

return arr

#Average will take an array of 5 classified DT data, and will set the value to whatever the

#majority is. Will only work with 5 classified DT's but can be tweaked to work for other values

def Average(arr):

for i in range(len(arr)):

if arr[i] >= 3:

arr[i] = 1

else:

arr[i] = 0

return arr

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

# Builds a one-node decision tree to classify data

# It tries n\_splits random splits and keeps the one that results in the lowest entropy

if n\_splits < 10:

n\_splits == 10

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)

for i in range(n\_splits):

while True:

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

#print("a", a)

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

#print("Ex,", ex)

thr = attributes[ex,a]

#print("Thr ", thr)

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

break

less = attributes[:,a] < thr

#print("less", len(less), less)

more = ~ less

#print("more", len(more), more)

#print("Target: ", len(target), target)

tgt\_less = target[less]

#print("tgt less",len(tgt\_less), tgt\_less)

tgt\_more = target[more]

#print("tgt more", len(tgt\_more), tgt\_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)

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

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

# build left decision subtree recursively

# does 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)

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:]]

#Constructing multiple decision trees to average them together to get more accurate data

#Try around 51?

'''

dt1 = buildDT(train\_data, train\_target, 100, 0.95, 25) #1000 200

dt2 = buildDT(train\_data, train\_target, 1000, 0.95, 200)

dt3 = buildDT(train\_data, train\_target, 1000, 0.95, 250)

dt4 = buildDT(train\_data, train\_target, 100, 0.95, 5)

dt5 = buildDT(train\_data, train\_target, 100, 0.95, 20)

'''

dt = []

for i in range(51):

dt.append(buildDT(train\_data, train\_target, 100, .95, 25))

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

#Classifying each tree and putting it in an array

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

for j in range(len(dt)):

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

#print("Train prediction predivide:", train\_pred[i], end = ' ')

train\_pred[i] = round(train\_pred[i] / len(dt))

#print("postdivide 2:", train\_pred[i])

#train\_pred = Average(train\_pred) #Average will make it so there are only 1's and 0s in the array

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)

#Same reason for train\_pred, classifying each tree and putting it in an array

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

for j in range(len(dt)):

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

#print("test prediction", test\_pred[i])

test\_pred[i] = round(test\_pred[i] / len(dt))

#test\_pred = Average(test\_pred) #Averaging each element of the arr to see if it's majority 1's or 0's

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

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

arr = np.zeros(10, dtype=int)

#Printing data for each tree

print('Number of nodes for dt[0]', NumNodes(dt[0]))

print('Height of dt[0]', Height(dt[0]))

print('times each attribute was used to split dt[0]', AttributeSplitBy(dt[0],arr))

Statement of Academic Honesty

“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 provided inappropriate assistance to any student in the class.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_