Assignment 3-CT475

Brendan Connolly 12370346-4BS2

My Algorithm

I chose to implement the ID3 decision tree algorithm. ID3 produces a tree which can be used to classify unseen samples. It works in the following way:

1. Pick out the attribute (A) that best splits the data*
2. Set up a tree (T) with A as the root node
3. Split the data into separate sub-sets using A
4. Run the algorithm recursively on all of the sub-sets (creating sub-trees) but remove A from the usable attributes
5. Connect the sub-trees to T

Stop, either when there are no more attributes to split by or when all of the samples in the sub-set have the same class

*This is done by seeing which attribute gives the best information gain when used to split the data. Information gain can be calculated relatively easily but I won't go into that here.

Main Design Considerations

The two major challenges for this implementation were (1) getting ID3 to handle continuously valued attributes and (2) getting ID3 to handle more than 2 classes when working with continuous attributes.

(1) ID3 can naturally handle discrete valued attributes. Simply split the data on each of the values of the attribute. However with a continuous valued attribute, this is obviously not possible. To get over this problem I introduced a threshold system. That is, instead of splitting data along discrete values of an attribute, choose thresholds at which to split the data. Thresholds can only exist at a boundary between classes so there should be much fewer threshold candidates than there are data samples. We pick the best threshold value for a given attribute by working out which one gives the best information gain when used to split the data. This can get quite computationally intensive on complicated data-sets so is a penalty for using ID3 with continuously valued attributes.

(2) ID3 can handle more than 2 attributes normally but with the threshold system it becomes tricky. Each extra class means we have to introduce an extra distinct threshold to split the data (E.g. with 3 classes we need to split the data along two thresholds rather than 1). Not only this, but we would need to keep track of which 2 classes each threshold was separating. Then we would need to test each combination of thresholds separately for information gain. This could quickly spiral out of control computationally so I chose another route round the problem.

For each class I made a separate decision tree, one that would distinguish between "class" and "notclass" (2 classes so only 1 threshold needed at each attribute split). Each decision that was made was given a certainty score. When it came to testing, each case was run through all of the trees. Each tree got a vote on the decision. The tree with the highest certainty score is the one that gets the casting vote.

Other Design Considerations

- Upon splitting the data sometimes one of the sub-data sets would end up with nothing in it. This could happen if I had only two samples with a different class but an identical splitting Attribute value (e.g. samples: (1) 'snowyOwl" body-width=6, (2) "notsnowyOwl" body-width=6, if we try to split on body-width, we won't be able to). It could also happen if the chosen threshold happened to be the maximum of the splitting attribute values. To deal with the first case I simply ignored situations by returning 0 information gain. In the second case I moved the threshold down slightly (to the next available number). In both cases some accuracy was traded for a program that actually worked!
- I made a mistake when beginning my implementation by referring to attributes and classes by their string names rather than just representative indices. I had an idea of writing some kind of visualization function for my models however it would have taken too long. Using the string names would have made this visualization function easier to write.
- I would have been better to use some kind of sparse matrix format to store my trees as they generally were made up of mostly 0's. If tested on a bigger data set my program may not be memory efficient.
- I had to come up with a system for giving certainty values for decisions in my trees. One possible way would be to just count the number of instances of the majority class in a Data set and leave that as the certainty score when ID3 terminates. This is a bit crude and could give false results if there is an almost even distribution of 2 classes in a data set. Another way of doing it would be to use the proportion of the majority class in a data set as the certainty score. This could mean that a Data set with only 1 instance in it would have the same certainty as a data set with 50 of the same class present. In the end I used a mixture; I multiplied the two discussed figures and returned that as the certainty score.

Layout of Input Data (The following are included as a warning when my code is run)

- The input data file must be a standard csv file with columns separated by commas and rows separated by new-line characters
- The first line of the data file has to contain attribute names that act as titles for their columns but the last column should not have a title
- The last column in the data file has to be the class attribute

My Results

All of the code I used to test my models is included in the appendix. The names of the relevant functions are: test_model(), use_model(), get_vote() and split_data_for _testing() .

My program performs 10 fold cross validation by default, here are the accuracy results for a sample run on owls15.csv:

Run 1: 91.11% Run 2: 88.89% Run 3: 95.56% Run 4: 77.78% Run 5: 91.11% Run 6: 88.89%
Run 7: 95.56% Run 8: 91.11% Run 9: 91.11% Run 10: 91.11%

Average accuracy: 90.22%

Note: On running my code you can get another set of accuracies and all predicted + actual results will be written to an output file of your choice.

Observations

An observation I would make is that the prediction accuracies seem to be suspiciously high. However having a look at the training data will explain this. The third column or 'body-width' of the samples is almost conclusive on splitting up the data. By this I mean any owls with 'body-width' < 2 are definitely long eared owls. No further analysis needed. Owls with 'body-width' > 5 are almost certainly snowy owls. Anything in between is almost certainly a barn owl. This means that ID3 will work really well on this data set but may not work as well on another more complex data set where boundaries are not so concrete. This can be shown by running the code on the illness data from Assignments 1 and 2 where my code will give about 75% accuracy.

My code obtains similar results (about 75ish% prediction accuracy as opposed to 77-78%) to Wekas implementation of this algorithm for the illness data (if at a slower pace). I take this as proof that my code, whilst perhaps lacking efficiency, is a fairly good implementation of the ID3 algorithm.

Conclusions

My implementation of the ID3 algorithm is fairly robust (as far as I can tell!) and performs reasonably well on both the owls data set and the illness data set.

Appendix – My Code

```
#Author: Brendan Connolly
#ID: 12370346
#Description:   This is a program that uses the ID3 classification algorithm to
build a classification tree from any user defined data set.
#               This implementation can deal with more than 2 classes and can also
handle numerical data attributes
#Date Modified: 11/11/15

#Note:  My comments will assume that anyone reading this code is already fimiliar
with the ID3 algorithm.
#       For info on ID3 visit: https://en.wikipedia.org/wiki/ID3_algorithm

import math
import operator
import random

#I assume that attributes only include the non class attributes (i.e Attributes
does not contain 'type')
def ID3(Data, Attributes, Classes):
    classList = get_all_classes(Data)   #classList is a copy of the 'type' column
from the Data passed in
    if classList.count(classList[0]) == len(classList): #check if all remaining
classes are the same
        vote = len(classList)   #The number of samples with the same class is a
measure of how sure we are of this classification
        return([[classList[0]+' '+str(vote)]])
    elif len(Attributes) == 0:
        majClass = find_maj_class(Data, Classes, classList)
        vote = classList.count(majClass)    #Similar to above the number of samples
of a majority class is a certainty value
        vote = vote * find_proportion(majClass, Data)   #However we need to temper
this certainty by mutipying by the (number of instances of the majority class) by
                                                        #(the proportion of the
majority class)
        return([[majClass+' '+str(vote)]])
    else:
        bestAttributeInd, threshold = find_best(Data, Attributes, Classes)
#bestAttrribute refers to the Attribute that gives the most information gain
        bestAttribute = Attributes[bestAttributeInd]
        Attributes.pop(bestAttributeInd)
        tree = [[bestAttribute]]    #We initialize tree to be a one node graph with
a label telling us what attribute it is splitting the data by.  This will get
combined with sub/supertrees later on.
        Data1, Data2 = [],[]
        for sample in Data:     #This for loop splits the data in a normal fashion
            if sample[bestAttributeInd] <= threshold:
                Data1.append(sample)
            else:
                Data2.append(sample)
        if len(Data2) == 0:     #Sometimes Data2 can end up with nothing in it(if
threshold happens to equal the max value of the bestAttribute column)
            for instance in Data1:  #So this for loop swaps a small amount of Data
into Data2(essentially moving the threshold down slightly)
                if instance[bestAttributeInd] == threshold:
                    temp = instance
                    Data1.remove(instance)
                    Data2.append(temp)
                else:
```

```
                    pass
        else:
            pass
        Data1 = remove_column(Data1, bestAttributeInd)  #This is necessary because
we need to make sure that the Data indices mirror the Attribute indices(which are
changing on every run)
        Data2 = remove_column(Data2, bestAttributeInd)
        subtree1 = ID3(Data1, Attributes, Classes)
        subtree2 = ID3(Data2, Attributes, Classes)
        tree = combine_trees(tree, subtree1)
        tree = combine_trees(tree, subtree2)
        tree[0][1] = threshold  #This line adds an edge(with a label of threshold)
between tree and subtree1
        tree[0][len(subtree1)+1] = threshold    #Edge between tree and subtree2
    return(tree)

def read_data(fileName):
    openFile = open(fileName, mode = 'r')
    inString = openFile.read()
    Data = [[]] #Data will be a list of lists
    i = 0 #i keeps track of where we are at in the file
    temp = ''
    firstLine = True
    for c in inString:  #This for loop adds characters from the file into temp
until it sees '\n' or ','. Then it adds temp to Data
        if firstLine == True:
            if c == '\n':
                firstLine = False
            else:
                pass
        else:
            if c == '\n':
                Data[i].append(temp)
                Data.append([])
                temp = ''
                i += 1
            elif c == ',':
                Data[i].append(temp)
                temp = ''
            else:
                temp = temp + c
    if Data[-1] == []:  #This condition means that this function can handle a new
line character at the end of the file
        Data[-2].append(temp)
        Data.pop()
    else:
        Data[-1].append(temp)
    for i in range(len(Data)):  #These for loops re-casts all of the numerical
strings in Data as actual flaoting point numbers
        for j in range(len(Data[0])-1):
            Data[i][j] = float(Data[i][j])
    openFile.close()
    return(Data)

#This function reads the Attributes from the top line of a csv file and returns
them
def get_attributes(fileName):
    openFile = open(fileName, mode = 'r')
    Attributes = []
```

```
    temp = ''
    line = openFile.readline()
    for c in line:  #This for loop adds characters from the file into temp until it
sees ','. Then it adds temp to Attributes
        if c == ',':
            Attributes.append(temp)
            temp = ''
        else:
            temp = temp + c
    openFile.close()
    return(Attributes)

#This function returns a list of all classes in the Data (used in main)
def get_classes(Data):
    Classes = []
    for i in range(len(Data)):
        curClass = Data[i][-1]
        if curClass not in Classes:
            Classes.append(curClass)
        else:
            pass
    return(Classes)

#Returns the most common class in the classes column
def find_maj_class(Data, Classes, classesColumn):
    maxc = 0
    for c in Classes:
        numOfClassc = classesColumn.count(c)
        if numOfClassc > maxc:
            maxc = numOfClassc
            curMax = c
        else:
            pass
    return(curMax)

#Returns the best attribute for splitting the Data and the threshold at which that
attribute should be split
def find_best(Data, Attributes, Classes):
    GainList = []
    ThreshList = []
    for i in range(len(Attributes)):    #This for loop computes the information
gain associated with slitting the data on each attribute
        threshold = find_threshold(i, Data, Classes)
        ThreshList.append(threshold)
        gain = Gain(Data, i, threshold, Classes)
        GainList.append(gain)
    maxi = max(GainList)                #We find the attribute with the best
information gain
    AttIndex = GainList.index(maxi)     #Finding the position of the maximum
element
    threshold = ThreshList[AttIndex]    #Retrieve the threshold associated with the
best attribute
    return(AttIndex, threshold)

#Finds the best threshold(information gain wise) and returns it
def find_threshold(AttributeIndex, Data, Classes):
    sortedData = sort_Data_by_column(Data, AttributeIndex)  #In order to find
posssible thresholds we have to have the Data sorted by AttributeIndex column
    allClasses = get_all_classes(sortedData)    #Gives a copy of the 'type' column
```

```
    candidates = []     #Will be list of the start value of every viable threshold
    for i in range(len(allClasses)-1):
        j = i + 1
        if allClasses[i] != allClasses[j]:  #Whenever, in the Data, the class of
one sample is different to the class of the next:
            candidates.append(sortedData[i][AttributeIndex])    #We save it as a
candidate threshold
        else:
            pass
    GainList = []
    for candidate in candidates:    #We find the info gain of each candidate
threshold
        GainList.append(Gain(sortedData, AttributeIndex, candidate, Classes))
    maxi = max(GainList)
    index = GainList.index(maxi)    #We return the threshold with the highest info
gain
    threshold = candidates[index]
    return(threshold)

#Returns the information gain obtained by splitting the Data by the specified
attribute at the specified threshold
def Gain(Data, AttributeIndex, threshold, Classes):
    Data1 = []
    Data2 = []
    workingColumn = []
    for point in Data:  #Produce a copy of the attribute column from the data
        workingColumn.append(point[AttributeIndex])
    if max(workingColumn) == threshold: #This conditional statement protects
agianst Data2 being empty(causes problems down the line with the entropy
calculation)
        for sample in Data: #Slitting the Data
            curNum = sample[AttributeIndex]
            if curNum >= threshold:
                Data1.append(sample)
            else:
                Data2.append(sample)
    else:
        for sample in Data: #Splitting the Data with a slightly lowered threshold
            curNum = sample[AttributeIndex]
            if curNum <= threshold:
                Data1.append(sample)
            else:
                Data2.append(sample)
    if len(Data1) == 0 or len(Data2) == 0:  #When the Data consists of only samples
where the values of the splitting Attribute are all the same it is impossible to
                                            #split the Data so we just ignore these
cases by saying that info gain is 0
        return(0)
    else:
        entSum = ((len(Data1)/len(Data)) * ent(Data1, Classes)) +
((len(Data2)/len(Data)) * ent(Data2, Classes))
        return(ent(Data, Classes) - entSum)

#Returns the proportion af a class in a certain Data set
def find_proportion(Class, Data):
    count_classes = get_all_classes(Data)
    num = count_classes.count(Class)
    return(num/len(count_classes))
```

```
#This function returns the class column from the Data passed in
def get_all_classes(Data):
    count_classes = []
    for sample in Data:
        count_classes.append(sample[-1])
    return(count_classes)

#Return the entropy of given Data set
def ent(Data, Classes):
    Ent = 0.0
    for c in Classes:
        propc = find_proportion(c, Data)
        if propc > 0:   #Making sure that we never take a log of 0
            Ent = Ent - (propc * math.log2(propc))
        else:
            pass
    return(Ent)

#Does what it says on the tin
def sort_Data_by_column(Data, AttributeIndex):
    return(sorted(Data, key=operator.itemgetter(AttributeIndex)))   #Sorted is a
default python function that creates a new sorted list from an old unsorted list
                                                                    #The itemgetter
part just fetches the right colunm to dort by

#Removes the column specified by AttributeIndex from the Data
def remove_column(Data, AttributeIndex):
    for sample in Data:
        sample = sample.pop(AttributeIndex)
    return(Data)

#Given a class C this function takes all instances in the data that are not C and
changes them into the string "notC"
def data_prep(Data, ClassIndex, Classes):
    replaceString = 'not' + Classes[ClassIndex]
    for i in range(len(Data)):
        if Data[i][-1] != Classes[ClassIndex]:
            Data[i][-1] = replaceString
        else:
            pass
    newClasses = [] #We no longer have the original classes we just have some class
C and notC we return newClasses to reflect this
    newClasses.append(Classes[ClassIndex])
    newClasses.append(replaceString)
    return(Data, newClasses)

#Returns a tree consisting of parTree and ChilTree combined
def combine_trees(parTree, chilTree):
    N1,N2 = len(parTree),len(chilTree)
    newTree = parTree   #Top left of matrix is just parTree
    for i in range(N2):
        newTree.append([0]) #Gives the number of nodes required to add chilTree to
newTree
        for j in range(N1-1):
            newTree[i+N1].append(0) #Fills in the bottom left corner of the Matrix
with 0's
        for k in range(N2):
            newTree[i+N1].append(chilTree[i][k])    #Fills the bottom left corner
with the contents of chilTree
```

```
    for i in range(N1):
        for j in range(N2):
            newTree[i].append(0)    #Fills the top right corner of the adjacency
matrix with 0's
    return(newTree)

#Returns training Data and testData where training Data is a random 2/3 of the
original Data set and testData makes up the other 1/3
def split_data_for_testing(trainingData):
    count = 0
    testData = []
    runUntil = int(1/3*len(trainingData))
    while count < runUntil:
        randNum = random.randint(0,len(trainingData)-1)
        sampleToMove = trainingData.pop(randNum)
        testData.append(sampleToMove)
        count += 1
    return(trainingData, testData)

#Takes in a list of results from the models and returns the result that we are most
sure about for a certain sample
def get_vote(possibilities):    #each element of possibilities is either 'class' or
'notclass' and has a certainty score associated with it
    votes = []
    notCount = 0
    for j in range(len(possibilities)): #This for loop finds out if all of the
models vote that the sample is 'notclass'
        if possibilities[j][:3] == "not":
            notCount += 1
        else:
            break
    if notCount < len(possibilities):   #If at least one of the decisions is
'class' rather than 'notclass' then we just need to find the highest certainty and
return that decision
        for i in range(len(possibilities)):
            votes.append(0)
            if possibilities[i][:3] == "not":   #If the first 3 letters of a result
are "not" then this won't be useful for classification and we can ignore it
                pass
            else:
                count = 0
                c = possibilities[i][count]
                while c !=  ' ':    #The certainty factor of a classification is
the (number of samples of a certain class in a Data set) * (proprotion of that
class in the Data set)
                                    #The classifications in the models of this
program have the format: "class certaintyFactor". Class is a string and certainty
factor is a float
                    count += 1      #This while loop finds the space in the
classification
                    c = possibilities[i][count]
                vote = float(possibilities[i][(count+1):])  #The certainty factor
is read and put into the votes list
                votes[i] = vote
        maxVote = max(votes)            #Then find the max vote and return the
class associated with it to the user
        maxIndex = votes.index(maxVote)
        return(possibilities[maxIndex])
```

```
    else:   #If all of the decisions are 'notclass' then we have to return the
decision that we are least sure of
        for i in range(len(possibilities)):
            votes.append(0)
            count = 0
            c = possibilities[i][count]
            while c !=  ' ':    #This while loop is the same as above
                count += 1
                c = possibilities[i][count]
            vote = float(possibilities[i][(count+1):])  #The certainty factor is
read and put into the votes list
            votes[i] = vote
        minVote = min(votes)    #Find minimum vote rather than maximum
        minIndex = votes.index(minVote)
        return(possibilities[minIndex][3:]) #[3:] gives 'class' out of 'notclass'.
This is done because we don't want to return 'notclass' since this doesn't exist in
the original data

#This fuction will take a sample and a set of decision trees and return what the
decision trees decide about the sample
#To understand this function you have to understand the structure of my generated
trees:
    #You start at (0,0) (my tree is in matrix form), this gives you an attribute
    #1. Then go right until you hit a float, this number will be the threshold by
which the attribute^ was split
    #2. All samples with value <= to the float^, for attribute^, continue downwards
in the matrix. Otherwise:
    #3. Any sample with value > float^, go right until you hit another float, then
proceed downwards in the matrix
        #When you go downwards you will either hit another attribute or a leaf node
(i.e a classification)
        #Stop if you hit a leaf
        #Repeat 1 to 3 above if you hit an attribute
def use_model(model, Sample, Attributes):
    posib = []
    for tree in model:
        i = 0
        j = 0
        curItem = tree[i][j]
        leaf = False
        while leaf == False:    #This while loop performs the procedure set out
above
            if curItem in Attributes:
                AttInd = Attributes.index(curItem)
                while type(curItem) != float:
                    j += 1
                    curItem = tree[i][j]
            elif type(curItem) == float:
                if Sample[AttInd] <= curItem:
                    pass
                else:
                    j += 1
                    curItem = tree[i][j]
                    while type(curItem) != float:
                        j += 1
                        curItem = tree[i][j]
                while type(curItem) != str:
                    i += 1
                    curItem = tree[i][j]
```

```
            else:
                posib.append(curItem)   #We will have the same number of trees as
classes so we add the individual results into posib
                leaf = True
    result = get_vote(posib)    #We then get a diffinitive judgement on the class
of a sample from get_vote()
    count = 0
    c = result[count]
    while c != ' ': #We only want to return the class, not the certainty factor
associated with that class
        count += 1
        c = result[count]
    return(result[:count])

#This function performs 10-fold cross validation on a data set and returns the
accuracies for each run
def test_model(Data, Classes, Attributes, fileName):
    outFile = open(fileName, 'w')
    accuracy = []
    for i in range(10):
        outString = 'Run: ' + str(i+1) + '\n'
        outFile.write(outString)
        newData = []
        for j in range(len(Data)):  #These for loops create an independant copy of
Data that can be modified without affecting Data
            newData.append([])
            for k in range(len(Data[0])):
                if type(Data[j][k]) == float:
                    thing = float(Data[j][k])
                else:
                    thing = ''.join(Data[j][k])
                newData[j].append(thing)
        newAttributes = []
        for j in range(len(Attributes)):  #Create an independant copy of Attributes
            newAttributes.append(0)
            thing = ''.join(Attributes[j])
            newAttributes[j] = thing
        trainingData, testData = split_data_for_testing(newData)    #Split the data
into training set and test set
        model = generate_model(Classes, trainingData, newAttributes)   #Performing
10 fold cross vaildation
        accNum = 0
        for sample in testData:
            result = use_model(model, sample, Attributes)
            outString = 'Predicted: ' + result + ' Actual: ' + sample[-1] + '\n'
            outFile.write(outString)
            if result == sample[-1]:
                accNum += 1     #If the model predicted the right classification
increase the accuracy numerator by 1
            else:
                pass
        accuracy.append(accNum/len(testData))
    return(accuracy)

#Returns a model based on some training data
def generate_model(Classes, Data, Attributes):
    trees = []
```

```
    for i in range(len(Classes)):   #I create the same number of trees as I have
classes (each tree can destinguish X from notX), we then face these trees off
against each other at alater stage
        newAttributes = []
        for j in range(len(Attributes)):  #Create an independant copy of Attributes
            newAttributes.append(0)
            thing = ''.join(Attributes[j])
            newAttributes[j] = thing
        newData = []
        for j in range(len(Data)):  #These for loops create an independant copy of
Data that can be modified without affecting Data
            newData.append([])
            for k in range(len(Data[0])):
                if type(Data[j][k]) == float:
                    thing = float(Data[j][k])
                else:
                    thing = ''.join(Data[j][k])
                newData[j].append(thing)
        workingData, workingClasses = data_prep(newData, i, Classes)
        newTree = ID3(workingData, newAttributes, Classes)
        trees.append(newTree)
    return(trees)   #Put all of the trees in a list and return them

def main():
    print("IMPORTANT: 1. The input file must be a csv")
    print("           2. The first line of the file must contain attribute names")
    print("           3. The 'class' column must be the last column in the file")
    print("           4. Do not include the name of the 'class' column in the
attributes names")
    inFileName = input('Please enter the input file name: ')
    outFileName = input('Please enter the name you wish the output file to have: ')
    Data = read_data(inFileName)
    Classes = get_classes(Data)
    Attributes = get_attributes(inFileName)
    accuracy = test_model(Data, Classes, Attributes, outFileName)
    print("Individual run accuracies: ")
    for j in range(len(accuracy)):
        print("{0:.2f}".format(accuracy[j]*100), "%", sep = '', end = ' ')
    print('')
    print("Average accuracy: ",
"{0:.2f}".format(((sum(accuracy))/(len(accuracy)))*100), "%", sep = '')

main()
```