

Mark Makris 19230705

Machine Learning CT4101

Assignment 2

Decision Tree Algorithm

This algorithm builds a tree of choices to categorize groups in order to make decisions. It is used to decide what to categorize something as or what decision to make based on given information. It is built by finding the attribute that has the most amount of information gain, branching from that and finding the next attribute with the highest information gain. It makes these decisions solely based on which attribute gives the most information and not how it interacts with any of the other given attribute. The basic sequence of events looks like this:

- Apply the information gain and entropy formula to every attribute
- Create a node with the attribute with the highest information gain
- Create branches from that node for each of its categories
- This gets repeated until each branch reaches a low enough entropy level or runs out of categories that provide enough information gain

Implementation Original

I chose to attempt to make a decision tree from scratch because I saw that it had potential from the good results in the first assignment when I used one. It is also a relatively straight forward model to create. My Model consisted of nodes and leaves. Each node consisted of an attribute to split on and the value of the attribute to split on, along with the two sides of the branch that the split created. The leaves consisted of a list of possible outcomes once it had reached that point. Each possible outcome had a probability and would randomly generate a number to choose an item to predict.

This algorithm reads in the text file of hazelnuts into a list of lists. Each list in the list contains all the attributes of a single hazelnut with the last one being its type. Then for 10 folds it randomly selects 2/3 of all the hazelnuts to create a training set and the other 1/3 becomes the testing set each time.

First the training set is sent to the splitter function to create the tree model of nodes and leaves. This creates a new list of lists without any of the types that holds all the unique attributes of the hazelnuts in the training data. From there it loops through every unique attribute and tests for information gain. This is done with a generic information gain formula that takes in the impurity of all the training data, the true side of the split, and the false side of the split. It looks for which attribute value has the greatest amount of purity and sees if that is any different from what is currently available.

Once the best value to split on has been chosen it will either create a node or leaf. If the information gain is below 14% it will create a new leaf. This was chosen as to attempt to prevent overfitting the data. If the gain is above 14% it will split the data into two creating a new node that calls the splitter function again with the true values and again for the false values, continuing until it ends in all leaves.

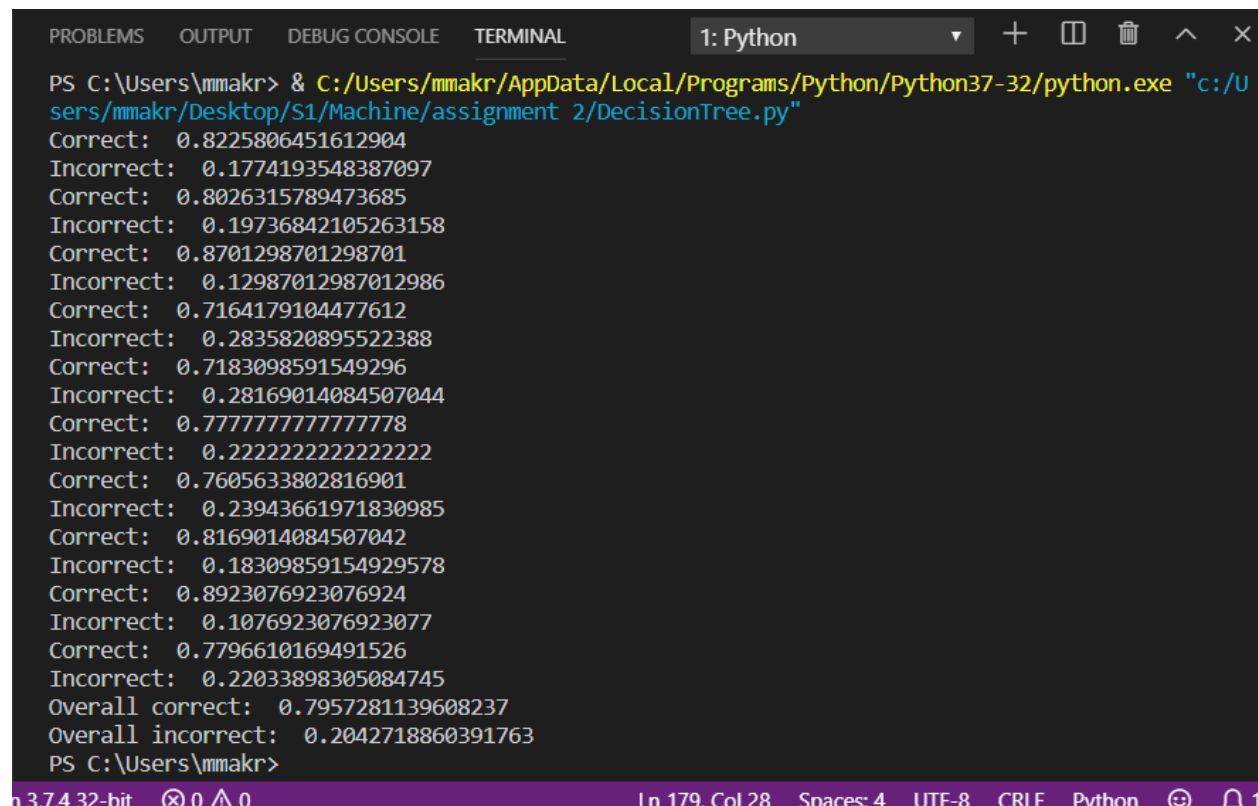
Finally, after the model is created for that fold it will take each item in the training data and predict the type using the model to see if it was correct or incorrect. It will return the percentage of total correctly predicted and incorrectly predicted. It does that for all 10 folds and then takes an average of all the accuracies and gives an overall percentage of incorrect and correct results.

Comparison library

For the comparison library I used my decision tree from assignment 1. This was made using Weka which was simple to use and produced great results. It just required a csv and could create the decision tree with the click of a button.

Test

Personal decision tree:



```
PS C:\Users\mmakr> & C:/Users/mmakr/AppData/Local/Programs/Python/Python37-32/python.exe "c:/Users/mmakr/Desktop/S1/Machine/assignment 2/DecisionTree.py"
Correct: 0.8225806451612904
Incorrect: 0.1774193548387097
Correct: 0.8026315789473685
Incorrect: 0.19736842105263158
Correct: 0.8701298701298701
Incorrect: 0.12987012987012986
Correct: 0.7164179104477612
Incorrect: 0.2835820895522388
Correct: 0.7183098591549296
Incorrect: 0.28169014084507044
Correct: 0.7777777777777778
Incorrect: 0.2222222222222222
Correct: 0.7605633802816901
Incorrect: 0.23943661971830985
Correct: 0.8169014084507042
Incorrect: 0.18309859154929578
Correct: 0.8923076923076924
Incorrect: 0.1076923076923077
Correct: 0.7796610169491526
Incorrect: 0.22033898305084745
Overall correct: 0.7957281139608237
Overall incorrect: 0.2042718860391763
PS C:\Users\mmakr>
```

The results of my decision tree were fair but not perfect. It correctly predicted about 80% of the hazelnuts after 10 folds with the best fold having a rate of 89% correct, and the worst having only 71% correct.

Weka decision tree:

```
=== Summary ===

Correctly Classified Instances      200          100      %
Incorrectly Classified Instances    0           0      %
Kappa statistic                     1
Mean absolute error                 0
Root mean squared error             0
Relative absolute error              0      %
Root relative squared error          0      %
Total Number of Instances          200

=== Detailed Accuracy By Class ===

      TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
      1.000    0.000    1.000    1.000    1.000    1.000    1.000    1.000    c_avellana
      1.000    0.000    1.000    1.000    1.000    1.000    1.000    1.000    c_americana
      1.000    0.000    1.000    1.000    1.000    1.000    1.000    1.000    c_cornuta
Weighted Avg.   1.000    0.000    1.000    1.000    1.000    1.000    1.000    1.000

=== Confusion Matrix ===

  a  b  c  <-- classified as
64  0  0 | a = c_avellana
 0 70  0 | b = c_americana
 0  0 66 | c = c_cornuta
```

The Weka decision tree was able to correctly predict all the Hazelnuts. This is somewhat to be expected from a fully developed program that is so widely used.

Conclusion/Observations

The algorithm I created performed about where I expected it to. Because this was the first classification algorithm, I have ever made it makes sense that it was not perfect. Room for improvement could be made in a few areas. Tweaking of how exactly the information gain was calculated could possibly be tweaked but deciding how to make the nodes and leaves is most likely where it opened up for allowing 20% incorrect predictions. Deciding to limit the information gain to prevent overfitting could be a reason it did not come out perfect like the Weka algorithm. I do not think there is any way to see what the Weka algorithm looks exactly like, but it would be interesting to compare the two to see where it made design decisions that allow it to be so accurate with the same data set. The variability of each fold was greater than I had expected with a range of about 20%. I would have expected something smaller like 10-15% variability on the amount correctly classified. The algorithm I created did not work as well as I knew a decision tree could because of the results I found from Weka but, overall it created a tree well and produced at least mostly correct answers.

[illegible]

```

        left = self.trueSide.__repr__()
        right = self.falseSide.__repr__()
        return "Attribute: " + labels[self.attr] + " <= " + self.split + "\tTrue:"
" + left + "False:" + right + "\n"

#leaf where the tree stops
class Leaf:
    def __init__(self, probs):
        self.probs = probs #attribute, value, total
    def getClass(self):
        return "Leaf"
    def getChoice(self):
        rand = random.uniform(0,1) #random number
        #find the item that it fits into
        for item in self.probs:
            if rand <= item[2]:
                return item[0]
    def __repr__(self):
        out = ""
        for item in self.probs:
            out += "attr: "
            out += item[0]
            out += " prob: "
            out += str(item[1])
            out += "%\n"
        return out

#TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL-
TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL-TREE MODEL

#CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL-
CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL

#loop through data
def splitter(training):
    options = []
    #for all variables that are not id or type
    for i in range(0,len(training[0:len(training[0])-1])):
        holder = []
        for row in training:
            if row[i] not in holder:
                holder.append(row[i])
        options.append(holder)

    #get unique labels
    uniqTypes = []
    for typ in training:

```

```

        if typ[-1] not in uniqTypes:
            uniqTypes.append(typ)

#get the amount of items of a certain type in a list
def counts(train):
    typeCounts = {}
    for row in train:
        if str(row) not in typeCounts:
            typeCounts[str(row)] = 1
        else:
            typeCounts[str(row)] += 1
    return typeCounts

#impurity function returns impurity
def impurity(train):
    typeCounts = counts(train)
    imp = 1
    for item in typeCounts:
        prob = float(typeCounts[item]) / len(train)
        imp -= prob**2
    return imp

#impurity
bestSplit = [0,0,0,0,0] #attribute, value, gains, impurity true, impurity false
se
##loop through all the possible questions
for i in range(0,len(options)): #i is an attribute position
    for j in range(0,len(options[i])): #j is row or an item position
        #per question
        trueType = []
        falseType = []
        allType = []
        #categorize each item type
        for row in range(0,len(training)):
            if (training[row][i] <= options[i][j]):
                trueType.append(training[row][-1])
            else:
                falseType.append(training[row][-1])
                allType.append(training[row][-1])
        #find impurity & gain
        impurityNode = impurity(allType)
        impurityT = impurity(trueType)
        impurityF = impurity(falseType)
        prob = float(len(trueType)) / (len(trueType)+len(falseType))
        gain = impurityNode - (prob * impurityT) - ((1-prob) * impurityF)

```

```

        if (gain > bestSplit[2]):
            bestSplit = [i, j, gain, impurityT, impurityF]

#start building tree
#makes a new node(false) or leaf(true)
if (bestSplit[2] < .14):
    leaves = []
    unique = []
    total = 0
    for row in training:
        if row[-1] not in unique:
            unique.append(row[-1])
    for key in unique:
        amt = 0 #amount of that key in data set
        for row in training:
            if row[-1] == key:
                amt += 1
        perc = amt/float(len(training))
        total += perc
        leaves.append([key, perc, total])
    return Leaf(leaves)
else:
    trueData = []
    falseData = []
    for row in training:
        F = float(row[bestSplit[0]])
        M = float(options[bestSplit[0]][bestSplit[1]])
        if (F<=M):
            trueData.append(row)
        else:
            falseData.append(row)
    return Node(bestSplit[0],options[bestSplit[0]][bestSplit[1]], splitter(tr
ueData), splitter(falseData))

#print("Tree: \n", Tree)
#CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL-
CREATING MODEL-CREATING MODEL-CREATING MODEL-CREATING MODEL

#TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL-
TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL
def test(testing, Tree):
    actuals = []
    predictions = []
    #create a list of the actual types
    for row in testing:

```

```

actuals.append(row[-1])

#predict item type
def predict(item, model):
    if model.getClass() == "Leaf":
        predictions.append(model.getChoice())
    elif item[model.attr] <= model.split:
        predict(item, model.trueSide)
    else:
        predict(item, model.falseSide)

#loop through test items
for item in testing:
    predict(item, Tree)

#accuracy of model
correct = 0
incorrect = 0
#compare actual to predicted type
for i in range(0, len(actuals)):
    if actuals[i] == predictions[i]:
        correct += 1
    else:
        incorrect += 1
print("Correct: ", correct / (correct+incorrect))
print("Incorrect: ", incorrect / (correct+incorrect))
return [correct/(correct+incorrect), incorrect/(correct+incorrect)]

#TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL-
TESTING MODEL-TESTING MODEL-TESTING MODEL-TESTING MODEL

#FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-
FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING

#seed for testing
random.seed(100)
overall = [0,0] #overall accuracies
#fold ten times
for i in range(0,10):
    #split the data
    training = []
    testing = []
    for i in data:
        rand = random.uniform(0,1)
        if rand <= .66:
            training.append(i)
        else:

```



```
        testing.append(i)
    Tree = splitter(training) #model create
    fold = test(testing, Tree) #test model
    overall[0] += fold[0]
    overall[1] += fold[1]
print("Overall correct: ", overall[0]/10)
print("Overall incorrect: ", overall[1]/10)
#FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-
FOLDING-FOLDING-FOLDING-FOLDING-FOLDING-FOLDING
```