Decision Tree Classifier using Information Gain and Entropy Calculations

In [304...
```
from sklearn import fetch_ucirepo
import pandas
import math
import numpy
import random

import copy

random.seed(1) # reproducibility

# fetch dataset
mushroom = fetch_ucirepo(id=73)

# data (as pandas dataframes)
X = mushroom.data.features
y = mushroom.data.targets

def subset(arr, feature, value):
    index = list(column_names).index(feature)
    return [item for item in arr if item[index] == value]


def split(arr, base = findroot(), unused = list(column_names), n =0, max_depth=4):
    if n >= max_depth:
        return
    feature = select_infogain(arr, unused, base)[0]
    node = [feature, []]
    base.append(node)
    unique = set([f[feature] ,values])
    l = unused.index(feature)
    new = unused[:l]+unused[l+1:]

    for value in unique:
        if not pandas.isnull(value):
            sub = subset(arr, feature, value)
            if select_infogain(sub, unused, new, base)[0]
                if new.entropy(sub)[2] == 0:
                    node[1].append([value, 'y'])
                else:
                    node[1].append([value, 'n'])
            else:
                node[1].append([value, split(sub, new, node[1], n+1, max_depth)])
                split(sub, new,ent(sub)[3], unused[:], n + 1, max_depth)


def new_ent(arr):
    e_count = 0
    p_count= 0
    for item in arr:
        if item[-1] == 'p':
            p_count+=1
        elif item[-1]=='e':
            e_count+=1
    total = e_count + p_count
    if e_count == 0 or p_count == 0:
        base_entropy = 0
    else:
        base_entropy = -1*((e_count/total)*math.log(e_count/total)) + ((p_count/total)*math.log(p_count/total)) )
    return [base_entropy, e_count, p_count]

def findroot(): #finds the root of the tree we will use
    values = y['poisonous'].values
    ecount = 0
    pcount = 0
    for v in values:
        if v == 'e':
            ecount+=1
        elif v== 'p':
            pcount+=1

    total = ecount + pcount
    base_entropy = -1*((ecount/total)*math.log(ecount/total)) + ((pcount/total)*math.log(pcount/total)) )

    return base_entropy
    # need to create ents which checks each feature then checks for all unique values
    #pass array down with the base entropy, and then check each columns unique values and then


def select_infogain(arr, unused_features, base):

    infogains = []
    for f in unused_features:
        values = []
        unique = set([list(X[f].values)])

        for char in unique:
            ngroup = n_group(arr[:], f, char)
            nentropy = new_entropy(arr[:], f, char)
            infogain = ngroup * char_entropy

        infogains.append(infogain)
    i = infogains.index(max(infogains))
    return [unused_features[i], max(infogains)]


# returns proportion of how many chars eg. a's in the category (column) eg. capsize array, a returns number of a's/total
def n_proportion(arr, column, value):
    index = list(column_names).index(column)
    count = 0
    for item in arr:
        if item[index] == value:
            count+=1
    if len(arr) ==0:
        return 0
    return count/len(arr)

def entropy(arr, column, value):
    poison = poison_prop(arr, column, value)
    not_poison = 1 - poison
    if poison == 0 or poison == 1:
        return 0
    if not_poison == 0 or not_poison == 1:
        return 0


    entropy = -1*((poison*math.log(poison))+(not_poison*math.log(not_poison)))

    return entropy


def poison_prop(arr, column, value): #returns proportion of poisonous when u pass a category and a value, eg. "capsize array","e"
    index = list(column_names).index(column)
    value_count = 0
    poison_count = 0

    for item in arr:
        if item[index] == value:
            value_count+=1
            if item[-1]=='p':
                poison_count+=1
    if value_count ==0:
        return 0
    return poison_count/value_count

def print_tree(t, depth=1):
    p = depth
    if isinstance(t, list)==True:
        for i in range(len(t)):
            item = t[i]
            if len(item)>2:
                print("  "*p + item[0:2])
            if isinstance(item[1],list)==True:
                print("  " + str(item[0]))
                print_tree(item[1], p+1)
            else:
                if item[1] == 'p':
                    print("  " * d + item[0]+ " --> " + "POISONOUS")
                elif item[1] == 'e':
                    print("  "*d + item[0] + " --> " + "EDIBLE")
                else:
                    print("  "*d + item[0] + " --> " + item[1])

column_names = X.columns.to_numpy()
classified_data = []
for i in range(len(X[feature_columns])):
    data = []
    for j in range(len(column_names)):
        data.append(X[column_names[j]].values[i])
    found = False

    for d in data:
        if pandas.isnull(d):
            found = True
    if found == False:
        data.append('poisonous'].values[i])

        classified_data.append(data)


# this puts the data in lists inside a list, classified data, with each mushroom's features in order.
```

In [305...
```
#Output for treedepth 2
tree = []
split(classified_data, findroot(), list(column_names), 0, 2)
print_tree(tree)
```

Out[305...
```
odor
  l --> POISONOUS
  c --> EDIBLE
  a --> EDIBLE
  p --> POISONOUS
  n --> spore-print-color
  f --> POISONOUS
  s --> POISONOUS
  m --> POISONOUS
  y --> POISONOUS
gill-size
  n --> EDIBLE
  b --> POISONOUS
spore-print-color
  k --> POISONOUS
  n --> POISONOUS
  h --> POISONOUS
  w --> cap-color
  r --> EDIBLE
  o --> POISONOUS
  b --> POISONOUS
  u --> POISONOUS
  y --> POISONOUS
```

In [306...
```
#Output for treedepth 3
tree = []
split(classified_data, findroot(), list(column_names), 0, 3)
print_tree(tree)
```

Out[306...
```
odor
  l --> POISONOUS
  c --> EDIBLE
  a --> EDIBLE
  p --> POISONOUS
  n --> spore-print-color
    p --> POISONOUS
    k --> POISONOUS
    n --> POISONOUS
    h --> POISONOUS
    w --> cap-color
    r --> EDIBLE
    o --> POISONOUS
    b --> POISONOUS
    u --> POISONOUS
    y --> POISONOUS
  f --> POISONOUS
  s --> POISONOUS
  m --> POISONOUS
  y --> POISONOUS
spore-print-color
  k --> POISONOUS
  n --> POISONOUS
  h --> POISONOUS
  w --> cap-color
  r --> EDIBLE
  o --> POISONOUS
  b --> POISONOUS
  u --> POISONOUS
  y --> POISONOUS
cap-color
  l --> POISONOUS
  g --> EDIBLE
  y --> POISONOUS
  w --> POISONOUS
  r --> EDIBLE
  b --> POISONOUS
  p --> EDIBLE
  u --> POISONOUS
  e --> POISONOUS
  c --> POISONOUS
```

In [307...
```
#Output for treedepth 4
tree = []
split(classified_data, findroot(), list(column_names), 0, 4)
print_tree(tree)
```

Out[307...
```
odor
  l --> POISONOUS
  c --> EDIBLE
  a --> EDIBLE
  p --> POISONOUS
  n --> spore-print-color
    p --> POISONOUS
    k --> POISONOUS
    n --> POISONOUS
    h --> POISONOUS
    w --> cap-color
    r --> EDIBLE
    o --> POISONOUS
    b --> POISONOUS
    u --> POISONOUS
    y --> POISONOUS
  f --> POISONOUS
  s --> POISONOUS
  m --> POISONOUS
  y --> POISONOUS
spore-print-color
  k --> POISONOUS
  n --> POISONOUS
  h --> POISONOUS
  w --> cap-color
  r --> EDIBLE
  o --> POISONOUS
  b --> POISONOUS
  u --> POISONOUS
  y --> POISONOUS
cap-color
  l --> POISONOUS
  g --> EDIBLE
  y --> POISONOUS
  w --> POISONOUS
  r --> EDIBLE
  b --> POISONOUS
  p --> EDIBLE
  u --> POISONOUS
  e --> POISONOUS
  c --> POISONOUS
```

In [308...
```
def newmush(): #makes new data for one mushroom in form same as classified data, features are randomized, can be anything.
    new_mushroom = []

    for feature in column_names:
        index = list(column_names).index(feature)
        unique = []
        for value in X:
            if not pandas.isnull(value):
                unique.append(value)
        i = random.randrange(0,len(unique))
        new_mushroom.append(unique[i])

    return new_mushroom

def testing(mushroom, full_tree, treedepth):
    tree = full_tree
    for item in full_tree:
        index = list(column_names).index(item[0])
        value = mushroom[index]
        items = item[1]
        for i in range(len(items)):
            if value == prediction[0]:
                if len(prediction) >=2:
                    if prediction[1] == 'p':
                        return "POISONOUS"
                    else:
                        return "EDIBLE"
                else:
                    if prediction[1] == 'p':
                        return "POISONOUS"
                    else:
                        return "EDIBLE"
            else:
                if prediction[2] == prediction[1]:
                    return "EDIBLE"
                else:
                    return "POISONOUS"


    count+=1
```

In [309...
```
#testing

"""
TESTING FOR NEW RANDOM DATA use this code instead of the values I provided below in lists (i,j,k,x)
i = newdata()
j = newdata()
k = newdata()
x = newdata()
"""


"""
run for visuals
print(column_names)
tree = []
split(classified_data, findroot(), list(column_names), 0, 4)
print_tree(tree)
print(i)
print(j)
print(k)
print(x)
"""

#test for specific data so that the results are reproducable.
i = ['l', 'y', 'w', 'f', 'a', 'c', 'b', 'k', 'e', 'c', 's', 's', 'w', 'w', 'p', 'w', 'o', 'e', 'n', 'a', 'g']
j = ['x', 'y', 'w', 'f', 'a', 'c', 'b', 'k', 'e', 'c', 's', 's', 'w', 'w', 'p', 'w', 'o', 'e', 'n', 'a', 'g']
x = ['f', 'y', 'y', 'f', 'a', 'f', 'b', 'k', 'e', 'c', 's', 's', 'w', 'w', 'p', 'w', 'o', 'e', 'n', 'a', 'g']
k = ['k', 'y', 'w', 'f', 'n', 'a', 'c', 'r', 'k', 'a', 'k', 's', 'g', 'g', 'p', 'w', 'o', 'e', 'w', 'v', 'l']



tree = []
split(classified_data, findroot(), list(column_names), 0, 4)
print(testing(i,tree,4))

tree = []
split(classified_data, findroot(), list(column_names), 0, 2)
print(testing(k,tree,2))

tree = []
split(classified_data, findroot(), list(column_names), 0, 3)
print(testing(j,tree,3))

tree = []
split(classified_data, findroot(), list(column_names), 0, 1)
print(testing(x,tree,1))
```

Out[309...
```
POISONOUS
POISONOUS
EDIBLE
EDIBLE
```

Question 1(c): Testing

I tested my algorithm of the decision tree by firstly making a method which constructs a new mushroom with data in the same formatting as my classified_data variable ['a','x',...'c'] etc. where each index corresponds to a column eg. 'a' corresponds to cap-size).

I randomised these using random.randrange and random.seed(1) to ensure reproducibility. With this new randomised mushroom I ran the method testing(mushroom, treedepth) which uses the tree made by the full dataset to make a prediction on a new mushroom. This is done by traversing through the tree until either reaching a leaf node or making a prediction based on the proportion of poisonous mushrooms and edible mushrooms in that leaf of the tree as to not exceed treedepth. When treedepth >= 4 there is a 100% accuracy as leave 4 contains solely of leaf nodes.

Random.seed(1) for reproducibility. However, code can be tested further by making more data. Random seed is set at the beginning of the main code (q1.ab) where libraries are imported and seed is initialised.

Make sure to set tree = [] if testing to get correct results.

for the sake of reproducibility i have used specific data. but using my newdata() function,

In [312...
```
def evaluation(full_set, full_tree, treedepth):

    correct = 0
    accuracy = 0
    total = 0

    for data in full_set:
        status = testing(data, full_tree, treedepth)
        if (data[-1] == 'p' and (status == "POISONOUS"):
            correct+=1
        elif (data[-1] == 'e') and (status == "EDIBLE"):
            correct+=1
        total+=1
    return correct/total
```

In [313...
```
full = []
for k in classified_data:
    full.append(k)

testing_data = []

for i in range(1000):
    random_n = random.randrange(0, len(full))
    testing_data.append(full[random_n])
    full.pop(random_n)


for j in range(1,5):
    tree = []
    split(full, findroot(), list(column_names), 0, 1)
    accuracy_1 = evaluation(testing_data, tree, 1)
    print("The testing accuracy for treedepth " + str(j) + " is " + str(accuracy_1))
```

```
The testing accuracy for treedepth 1 is 0.9633333333333333
The testing accuracy for treedepth 2 is 0.9966666666666667
The testing accuracy for treedepth 3 is 1.0
The testing accuracy for treedepth 4 is 1.0
```

Full accuracy when tree_depth >=3 because I removed the missing values from the dataset. Had I not done this the accuracy would only be 100% when tree_depth>=6.

I implemented a function which evaluates the whole decision tree procedure. First it forms two sets based on the full data set randomly one being the testing_data set and one being the training data set (full). It uses the training data set to form a decision tree that is split based on information gain and entropy.

Once the tree is formed we can test the accuracy of the data by using the test function 1 made in Q1(c) on each list of data. This will give us a prediction on whether the mushroom is poisonous or not and we can compare this to the actual data of the mushroom to see if the prediction was correct. Doing this for many mushroomsMake gives us an accuracy score. (number of prediction correct/total predictions).

For treedepth 1 the accuracy is 0.96425 for treedepth 2 the accuracy is 0.99725 For treedepth 4 it is 1.0 and if treedepth >=3 then the accuracy will be 1.0 (100%).

There are several other ways the decision tree can be implemented using different splitting criteria. A common splitting criteria is the gini impurity splitting criteria. Altering the splitting criteria will change how the decision tree determines which feature to split into at what level of the tree, thus resulting in different partitions and a completely different tree and thus with a different accuracy.

Gini impurity is measured as a number from 0 to 1 where 0 indicates all elements are part of the same class and 1 indicating elements are randomly amongst a variety of classes. 0.5 shows a normal distribution of the elements amongst the classes.

The gini impurity can be calculated by calculating gini coefficients for the subnodes of a node and then calculating the impurity of each node by calculating the weighted average of the distribution amongst the subnodes in the non-binary tree. The idea here is to minimise the number of incorrect nodes when classifying the mushrooms.

The gini impurity splitting criteria is based around the probability of data being misclassified. The entropy, information gain splitting criteria both focus on the impurity in a dataset. Gini entropy has a more favourable computing power meaning it is preferable when datasets are larger as the entropy splitting criteria will be more inefficient. However, using entropy typically takes longer but is also more accurate as it seeks the maximum information gain and the least entropy and minimises inaccuracy. For larger datasets, gini impurity is favourable due to the computing power. Additionally gini impurity is favourable for datasets where the classes are relatively balanced, but using the information gain is favourable when classes are more randomly distributed

Using gini impurity and entropy will usually give two different decision trees, which is important to note. This is because the splitting criteries are different which means the nodes selected are likely different. This means the decision tree will usually have a different accuracy depending on whether you use entropy/information gain or gini impurity to implement the algorithm.

In [315...
```
training_data = []
for k in classified_data:
    training_data.append(k)

for i in range(1,5):
    tree = []
    split(training_data, findroot(), list(column_names), 0, 1)
    accuracy_1 = evaluation(training_data, tree, 1)
    print("The training accuracy for treedepth " + str(i) + " is " + str(accuracy_1))
```

```
The training accuracy for treedepth 1 is 0.9644082211197732
The training accuracy for treedepth 2 is 0.9971651511112656
The training accuracy for treedepth 3 is 1.0
The training accuracy for treedepth 4 is 1.0
```

My evaluation can explain whether my tree is overfitting by now testing the training accuracy.

The training accuracy for treedepth 1 is 0.9644082211197732 The training accuracy for treedepth 2 is 0.9971651511112656 The training accuracy for treedepth 3 is 1.0 The training accuracy for treedepth 4 is 1.0 and we know from before The testing accuracy for treedepth 1 is 0.9633333333333333 The testing accuracy for treedepth 2 is 0.9966666666666667 The testing accuracy for treedepth 3 is 1.0 The testing accuracy for treedepth 4 is 1.0

Underfitting data is typically when training accuracy is low and it is usually overfitting when testing accuracy is low. In this case both the training accuracy and testing accuracy are high which means it is neither underfitting nor overfitting!