Project 2

March 19, 2020

Project 2

Below is the code for project 2, with output. In the code for building a single tree, I found direction from here. However, I tried to make enough changes to call it my own work, including using the gini method for splitting nodes. I found that GINI provided slightly more variety in the types of trees my ensemble method produced than did entropy. What follows directly is the function calls for building a decision tree and making predictions with it.

[9]:
```
import subprocess as sp
import pandas as pd
import numpy as np

sp.call('clear', shell = True)

raw_data = pd.read_csv('Data.csv').set_index('Instance')
data_train = raw_data.iloc[0:10,:]
data_val = raw_data.iloc[10:15,:]
del(raw_data)

#Calculate the GINI of the parent node
def parent_gini(df):
    #Isolate the class labels of the dataframe
    Class = df.keys()[-1]
    gini = 1
    #Create an iterable for the class labels
    targets = df[Class].unique()

    for target in targets:
        probability = df[Class].value_counts()[target]/len(df[Class])
        gini += -1*np.square(probability)

    return gini

#Calculate the GINI of each potential child node to find the maximum gain.
def child_gini(df,attribute):
    #As before, only this time we're going to iterate over the instance labels␣
 ↪and the class labels.
    Class = df.keys()[-1]
```

```
    targets = df[Class].unique()
    variables = df[attribute].unique()
    #This variable will become the weighted average at the end of the function
    gini_child = 0

    for variable in variables:
        gini_individual = 1

        for target in targets:
            numerator = len(df[attribute][df[attribute] == variable][df[Class]
 ↪== target])
            denominator = len(df[attribute][df[attribute] == variable])
            probability_individual = numerator/(denominator + epsilon)
            gini_individual += - np.square(probability_individual)
        #Calculate the GINI for each child and store it within the weighted
 ↪average
        probability_child = denominator/len(df)
        #Weighted Average
        gini_child += probability_child * gini_individual

    return gini_child

#Determine which child provides the maximum gain and return that to the build
 ↪function
def choose_node(df):
    information_gain = []

    for key in df.keys()[:-1]:
        information_gain.append(parent_gini(df) - child_gini(df,key))

    return df.keys()[:-1][np.argmax(information_gain)]

#Create a new dataframe which is a subset of the original and contains only the
 ↪data we will need at the next split node.
def build_subtree(df,node,value):
    return df[df[node] == value].reset_index(drop = True)

#Build the tree recursively
def build_decision_tree(df,tree = None):
    node = choose_node(df)
    attribute_values = np.unique(df[node])

    #Create a new subtree with node as the root
    if tree is None:
        tree = {}
        tree[node] = {}
```

```
    for value in attribute_values:
        subtree = build_subtree(df,node,value)
        class_values,counts = np.unique(subtree['Class'],return_counts = True)

        #If all the class labels are the same, create a leaf node, otherwise␣
 ↪split the node by calling the build function. This is the only
        #stopping condition I have, since the small number of attributes will␣
 ↪keep the tree rather short.
        if len(counts) == 1:
            tree[node][value] = class_values[0]
        else:
            tree[node][value] = build_decision_tree(subtree)

    return tree

#Make a prediction using our built tree and a new instance. This is in fact␣
 ↪just copied from the link I provided earlier. I couldn't make it
#different enough to call it mine, so I left it.
def predict(inst,tree):

    for nodes in tree.keys():

        value = inst[nodes]
        tree = tree[nodes][value]
        prediction = 0

        if type(tree) is dict:
            prediction = predict(inst, tree)
        else:
            prediction = tree
            break;

    return prediction
```

Below is the bagging portion of the algorithm. I've done work with a bootstrap algorithm before, and know that I could have created up to 10^{10} new datasets from the original training data. However, since the attribute B is evenly split 5:5, the tree building function was heavily biased towards beginning every tree with B as the root. This kept reproducing the two trees with B as the root which you can see below. From this, I chose to keep the number of bagging rounds to be quite small.

Using GINI did produce some more variety in the trees, as you can see below. When I was using entropy, that variety was non-existent and I only prduced the two trees mentioned above.

[31]:
```
from pprint import pprint

bagging_ensemble = {}
BOOTSTRAP = 10
```

```
for boot in range(1,BOOTSTRAP+1):
    new_data_train = data_train.sample(len(data_train), replace = True).
 ↪reset_index()
    new_data_train = new_data_train.drop(columns = ['Instance'])
    bagging_ensemble[boot] = {}
    tree = build_decision_tree(new_data_train)
    bagging_ensemble[boot] = tree

pprint(bagging_ensemble, width = 1)

final_predictions = []
for index in data_val.index:
    count = 0
    for boot in range(1,BOOTSTRAP+1):
        prediction = predict(data_val.loc[index,:],bagging_ensemble[boot])
        count += prediction
    #print(index, ": ", count)
    if count <= 0:
        final_predictions.append(-1)
    else:
        final_predictions.append(1)

print("Outcomes for the Validation Set: \n")
for index in data_val.index:
    if data_val.loc[index,'Class'] == final_predictions[index - 11]:
        print(index, ": ", final_predictions[index - 11], "Classified")
    else:
        print(index, ": ", final_predictions[index - 11], "Misclassified")
```

```
{1: {'A': {0: 1,
           1: {'B': {0: 1,
                     1: -1}}}},
 2: {'B': {0: 1,
           1: -1}},
 3: {'B': {0: 1,
           1: {'A': {0: {'C': {0: 1,
                               1: -1}},
                     1: -1}}}},
 4: {'B': {0: 1,
           1: {'A': {0: 1,
                     1: -1}}}},
 5: {'A': {0: {'C': {0: 1,
                     1: -1}},
           1: -1}},
 6: {'A': {0: {'B': {0: 1,
                     1: {'C': {0: 1,
                               1: -1}}}},
```

```
           1: -1}},
 7: {'B': {0: 1,
           1: {'A': {0: {'C': {0: 1,
                               1: -1}},
                     1: -1}}}},
 8: {'B': {0: 1,
           1: -1}},
 9: {'C': {0: {'A': {0: 1,
                     1: -1}},
           1: -1}},
 10: {'B': {0: 1,
            1: {'A': {0: {'C': {0: 1,
                                1: -1}},
                      1: -1}}}}}
Outcomes for the Validation Set:

11 :  1 Classified
12 :  -1 Misclassified
13 :  -1 Misclassified
14 :  1 Misclassified
15 :  1 Classified
```

Changing the number of bagging rounds would not improve the poor accuracy the model had on the validation dataset. Upon looking at the data more closely, the three instances which kept getting misclassified: 12, 13, 14, are each a direct contradiction of instances 4, 7, and 8 in the training data. Since the dataset was so small, there's no way the algorithm can learn these instances to classify them correctly.

[35]:
```
display("Training Data:", data_train, "Validation Data:", data_val)
```

```
'Training Data:'
```

Instance	A	B	C	Class
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	0	1	1	-1
5	1	0	0	1
6	1	0	0	1
7	1	1	0	-1
8	1	0	1	1
9	1	1	0	-1
10	1	1	0	-1

```
'Validation Data:'
```

```
           A  B  C  Class
Instance
11         0  0  0      1
12         0  1  1      1
13         1  1  0      1
14         1  0  1     -1
15         1  0  0      1
```

[ ]: