Assignment 4:

Arun Agarwal

915774866

Homework 4

2/12/2022 - 2/16/2022

In [1]:
```python
#Imports:
import pandas as pd
import numpy as np
import matplotlib as mp #For Visualization
```

Problem 1:

1. Develop a decision tree classification software on your own and apply it to develop a classification model for Donor dataset from Homework 3. Since this dataset is small, apply leave-one-out training and testing framework and report your findings.
2. Now apply this classifier to solve the Mushroom problem defined at https://www.kaggle.com/uciml/mushroom-classification (Links to an external site.). Evaluate results using 10-fold cross-validation and report your findings.
3. Compare test set accuracy when training a mushroom classifier using 500 vs. 5,000 training examples.
4. Report accuracy on training and test data when using a decision tree that has 10 vs 30 leaves.
5. Report attribute tests used in the decision tree with 10 leaves.
6. For a decision tree with 10 leaves report the number of positive and negative training examples at each internal node and at leaves.

a) Develop a decision tree classification software on your own and apply it to develop a classification model for Donor dataset from Homework 3. Since this dataset is small, apply leave-one-out training and testing framework and report your findings.

For this question, I will use an ID3 algorithm with a binary split to develop a classification model for the Donor dataset, shown below:

Name	Age	Salary	Donor
Nancy	21	37,000	N
Jim	27	41,000	N
Allen	43	61,000	Y
Jane	38	55,000	N
Steve	44	30,000	N
Peter	51	56,000	Y
Sayani	53	70,000	Y
Lata	56	74,000	Y
Mary	59	25,000	N
Victor	61	68,000	Y
Dale	63	51,000	Y

In [2]:
```python
#Reading in the Data:
df = pd.read_csv('donorData.txt', delimiter = ' ', header = 0)
df['Donor'] = df['Donor'].apply(lambda x: 1 if x == 'Y' else 0)
df = df.reset_index().drop(columns = ['Name'])
df
```

Out[2]:

	index	Age	Salary	Donor
0	0	21	37000	0
1	1	27	41000	0
2	2	43	61000	1
3	3	38	55000	0
4	4	44	30000	0
5	5	51	56000	1
6	6	53	70000	1
7	7	56	74000	1
8	8	59	25000	0
9	9	61	68000	1
10	10	63	51000	1

As stated in the textbook,

$$Entropy = -\sum_{i=0}^{c-1} p_i(t)(log_2 p_i(t))$$

, where $p_i(t)$ is the relative frequency of training instances that belong to class i at node t, $0 log_2 0 = 0$ in entropy calculations, and c is the total number of classes.

In our case, $c = 2$, $p_i(t)^2$ is the relative frequency for the two classes (Y and N), and the nodes t are Y and N.

Now, computing the entropy index for the entire Donor data set, with respect to the two classes, we have:

$$Entropy_{2class} = -\sum_{i=0}^{c-1} p_i(t)(log_2 p_i(t)) = -(\frac{6}{11})(log_2(\frac{6}{11})) - (\frac{5}{11})(log_2(\frac{5}{11})) \approx 0.994$$

In [3]:
```python
#We will next calculate the entropy of the entire dataset:
def calc_total_entropy(train_data, label, class_list):
    total_row = train_data.shape[0] #the total size of the dataset
    total_entr = 0

    for c in class_list: #for each class in the label
        total_class_count = train_data[train_data[label] == c].shape[0] #number of the class
        total_class_entr = - (total_class_count/total_row)*np.log2(total_class_count/total_row) #entropy of the
        total_entr += total_class_entr #adding the class entropy to the total entropy of the dataset

    return total_entr
```

In [4]:
```python
#We will next calculate the entropy for the filtered dataset:
def calc_entropy(feature_value_data, label, class_list):
    class_count = feature_value_data.shape[0]
    entropy = 0

    for c in class_list:
        label_class_count = feature_value_data[feature_value_data[label] == c].shape[0] #row count of class c
        entropy_class = 0
        if label_class_count != 0:
            probability_class = label_class_count/class_count #probability of the class
            entropy_class = - probability_class * np.log2(probability_class)  #entropy
        entropy += entropy_class
    return entropy
```

In [5]:
```python
#Next, we calculate the information gain for a feature.
def calc_info_gain(feature_name, train_data, label, class_list):
    feature_value_list = train_data[feature_name].unique() #unqiue values of the feature
    total_row = train_data.shape[0]
    feature_info = 0.0

    for feature_value in feature_value_list:
        feature_value_data = train_data[train_data[feature_name] == feature_value] #filtering rows with that fe
        feature_value_count = feature_value_data.shape[0]
        feature_value_entropy = calc_entropy(feature_value_data, label, class_list) #calculcating entropy for t
        feature_value_probability = feature_value_count/total_row
        feature_info += feature_value_probability * feature_value_entropy #calculating information of the featu

    return calc_total_entropy(train_data, label, class_list) - feature_info #calculating information gain by su
```

In [6]:
```python
#We now need to determine the most informative feature, or the one with the highest information gain:
def find_most_informative_feature(train_data, label, class_list):
    feature_list = train_data.columns.drop(label) #finding the feature names in the dataset
                                            #N.B. label is not a feature, so dropping it
    max_info_gain = -1
    max_info_feature = None

    for feature in feature_list:  #for each feature in the dataset
        feature_info_gain = calc_info_gain(feature, train_data, label, class_list)
        if max_info_gain < feature_info_gain: #selecting feature name with highest information gain
            max_info_gain = feature_info_gain
            max_info_feature = feature

    return max_info_feature
```

In [7]:
```python
#Moving on, we add a node to the tree. Since we have found the feature
#name with the highest information gain, we now need to generate a node
#in a tree and its value as a branch.
#If any of the features represent only one class, then we can say that
#the feature value represents a pure class. Otherwise, we have to extend
#it further. After selecting a pure class, we have to remove the rows
#from the dataset corresponding to the feature value.

def generate_sub_tree(feature_name, train_data, label, class_list):
    feature_value_count_dict = train_data[feature_name].value_counts(sort=False) #dictionary of the count of un
    tree = {} #sub tree or node

    for feature_value, count in feature_value_count_dict.iteritems():
        feature_value_data = train_data[train_data[feature_name] == feature_value] #dataset with only feature_n

        assigned_to_node = False #flag for tracking feature_value is pure class or not
        for c in class_list: #for each class
            class_count = feature_value_data[feature_value_data[label] == c].shape[0] #count of class c

            if class_count == count: #count of feature_value = count of class (pure class)
                tree[feature_value] = c #adding node to the tree
                train_data = train_data[train_data[feature_name] != feature_value] #removing rows with feature_
                assigned_to_node = True
        if not assigned_to_node: #not pure class
            tree[feature_value] = "?" #should extend the node, so the branch is marked with ?

    return tree, train_data
```

In [8]:
```python
#To perform the ID3 Algorithm, we will complete steps 4 through 8
#recursively. Thus, the overall steps are: finding the most informative
#feature, making a tree node with a feature name and feature values
#as branches (if pure class, adding lead node (= Class) to the tree node;
#if impure class, adding an expandable node to the tree node), shrinking
#updating the dataset according to the pure class, adding the node with
#branches into the tree, and expanding the branch of the next impure class
#with an updated dataset. The recursion will end when the dataset becomes
#empty after updating or when there is no expandable branch (all pure class).

def make_tree(root, prev_feature_value, train_data, label, class_list):
    if train_data.shape[0] != 0: #if dataset becomes empty after updating
        max_info_feature = find_most_informative_feature(train_data, label, class_list) #most informative featu
        tree, train_data = generate_sub_tree(max_info_feature, train_data, label, class_list) #getting tree nod
        next_root = None

        if prev_feature_value != None: #add to intermediate node of the tree
            root[prev_feature_value] = dict()
            root[prev_feature_value][max_info_feature] = tree
            next_root = root[prev_feature_value][max_info_feature]
        else: #add to root of the tree
            root[max_info_feature] = tree
            next_root = root[max_info_feature]

        for node, branch in list(next_root.items()): #iterating the tree node
            if branch == "?": #if it is expandable
                feature_value_data = train_data[train_data[max_info_feature] == node] #using the updated datase
                make_tree(next_root, node, feature_value_data, label, class_list) #recursive call with updated
```

In [9]:
```python
#We can start calling the recursive tree building algorithm of ID3 after
#finding the class names:
def id3(train_data_m, label):
    train_data = train_data_m.copy() #getting a copy of the dataset
    tree = {} #tree which will be updated
    class_list = train_data[label].unique() #getting unqiue classes of the label
    make_tree(tree, None, train_data_m, label, class_list) #start calling recursion
    return tree
```

In [10]:
```python
tree = id3(df, 'Donor')
```

In [11]:
```python
print(tree)
```

```
{'index': {0: 0, 1: 0, 2: 1, 3: 0, 4: 0, 5: 1, 6: 1, 7: 1, 8: 0, 9: 1, 10: 1}}
```

In [12]:
```python
#We not can use our tree for prediction. We recursively traverse the
#nested dictionary until an leaf node is found.

def predict(tree, instance):
    if not isinstance(tree, dict): #if it is leaf node
        return tree #return the value
    else:
        root_node = next(iter(tree)) #getting first key/feature name of the dictionary
        feature_value = instance[root_node] #value of the feature
        if feature_value in tree[root_node]: #checking the feature value in current tree node
            return predict(tree[root_node][feature_value], instance) #goto next feature
        else:
            return None
```

In [13]:
```python
#we evaluate the model. After predicting, we can calculate the difference
#of actual and predicted value in percentage.
def evaluate(tree, test_data_m, label):
    correct_preditct = 0
    wrong_preditct = 0
    for index, row in test_data_m.iterrows(): #for each row in the dataset
        result = predict(tree, test_data_m.iloc[index]) #predict the row
        if result == test_data_m[label].iloc[index]: #predicted value and expected value is same or not
            correct_preditct += 1 #increase correct count
        else:
            wrong_preditct += 1 #increase incorrect count
    accuracy = correct_preditct / (correct_preditct + wrong_preditct) #calculating accuracy
    return accuracy
```

In [14]:
```python
#It was requested that we use leave-one-out-cross validation to test our model. Since I could not figure out ho
#write the partitions without using the sklearn built-in leave-one-out classifier, I created the datasets by ha
df_0_9 = df[0:10]
df_0_9Test = df[10:].reset_index(drop = True)

df_1_10 = df.loc[[1,2,3,4,5,6,7,8,9,10],:]
df_1_10Test = df[:1].reset_index(drop=True)

df_2_0 = df.loc[[0,2,3,4,5,6,7,8,9,10],:]
df_2_0Test = df[1:2].reset_index(drop=True)

df_3_1 = df.loc[[0,1,3,4,5,6,7,8,9,10],:]
df_3_1Test = df[2:3].reset_index(drop=True)

df_4_2 = df.loc[[0,1,2,4,5,6,7,8,9,10],:]
df_4_2Test = df[3:4].reset_index(drop=True)

df_5_3 = df.loc[[0,1,2,3,5,6,7,8,9,10],:]
df_5_3Test = df[4:5].reset_index(drop=True)

df_6_4 = df.loc[[0,1,2,3,4,6,7,8,9,10],:]
df_6_4Test = df[5:6].reset_index(drop=True)

df_7_5 = df.loc[[0,1,2,3,4,5,7,8,9,10],:]
df_7_5Test = df[6:7].reset_index(drop=True)

df_8_6 = df.loc[[0,1,2,3,4,5,6,8,9,10],:]
df_8_6Test = df[7:8].reset_index(drop=True)

df_9_7 = df.loc[[0,1,2,3,4,5,6,7,9,10],:]
df_9_7Test = df[8:9].reset_index(drop=True)

df_10_8 = df.loc[[0,1,2,3,4,5,6,7,8,10],:]
df_10_8Test = df[9:10].reset_index(drop=True)
```

In [15]:
```python
#Making the trees for each of the training datasets
tree_0_9 = id3(df_0_9, 'Donor')
tree_1_10 = id3(df_1_10, 'Donor')
tree_2_0 = id3(df_2_0, 'Donor')
tree_3_1 = id3(df_3_1, 'Donor')
tree_4_2 = id3(df_4_2, 'Donor')
tree_5_3 = id3(df_5_3, 'Donor')
tree_6_4 = id3(df_6_4, 'Donor')
tree_7_5 = id3(df_7_5, 'Donor')
tree_8_6 = id3(df_8_6, 'Donor')
tree_9_7 = id3(df_9_7, 'Donor')
tree_10_8 = id3(df_10_8, 'Donor')
```

In [16]:
```python
#Evaluating on the testing dataset (the one instance that was not included) and printing out the accuracies:
accuracy_0_9 = evaluate(tree_0_9, df_0_9Test, 'Donor') #evaluating the test dataset
print(accuracy_0_9)

accuracy_1_10 = evaluate(tree_1_10, df_1_10Test, 'Donor') #evaluating the test dataset
print(accuracy_1_10)

accuracy_2_0 = evaluate(tree_2_0, df_2_0Test, 'Donor') #evaluating the test dataset
print(accuracy_2_0)

accuracy_3_1 = evaluate(tree_3_1, df_3_1Test, 'Donor') #evaluating the test dataset
print(accuracy_3_1)

accuracy_4_2 = evaluate(tree_4_2, df_4_2Test, 'Donor') #evaluating the test dataset
print(accuracy_4_2)

accuracy_5_3 = evaluate(tree_5_3, df_5_3Test, 'Donor') #evaluating the test dataset
print(accuracy_5_3)

accuracy_6_4 = evaluate(tree_6_4, df_6_4Test, 'Donor') #evaluating the test dataset
print(accuracy_6_4)

accuracy_7_5 = evaluate(tree_7_5, df_7_5Test, 'Donor') #evaluating the test dataset
print(accuracy_7_5)

accuracy_8_6 = evaluate(tree_8_6, df_8_6Test, 'Donor') #evaluating the test dataset
print(accuracy_8_6)

accuracy_9_7 = evaluate(tree_9_7, df_9_7Test, 'Donor') #evaluating the test dataset
print(accuracy_9_7)

accuracy_10_8 = evaluate(tree_10_8, df_10_8Test, 'Donor') #evaluating the test dataset
print(accuracy_10_8)

average_accuracy = (accuracy_0_9 + accuracy_1_10 + accuracy_2_0 + accuracy_3_1 + accuracy_4_2 + accuracy_5_3 +
print("Average Accuracy is", average_accuracy)
```

```
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
0.0
Average Accuracy is 0.0
```

From my experience, I have always been able to use sklearn to create models such as these. If the professor could have provided us with some assistance in writing such a model from scratch, especially for undergraduate students like myself, then this would have been more doable. I tried following an online tutorial to make the Decision Tree Classifier because I have never been taught how to make one (not in this class or any other class), but it appears that there may have been something wrong with the classifier. An accuracy percentage of 0 for all 11 cases is unlikely. Overall, we can say that the model did extremely poorly in classifying this dataset. At the same time, we see that for the mushroom dataset below, the model did well.

To understand how well our model should have performed, I have written the code using sklearn's decision tree classifier and leave one out model selector below:

In [17]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_val_score
```

In [18]:
```python
#define predictor and response variables
X = df[['index', 'Age', 'Salary']]
y = df['Donor']

#define cross-validation method to use
cv = LeaveOneOut()

#build multiple linear regression model
model = DecisionTreeClassifier()

#use LOOCV to evaluate model
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error',
                         cv=cv, n_jobs=-1)

#view mean absolute error
np.mean(np.absolute(scores))
```

Out[18]: 0.36363636363636365

This error, while significant, suggests that sklearn still performed better than our self-built model.

b) Now apply this classifier to solve the Mushroom problem defined at https://www.kaggle.com/uciml/mushroom-classification (Links to an external site.). Evaluate results using 10-fold cross-validation and report your findings.

In [19]:
```python
#Importing dataset:
dataset = pd.read_csv('mushrooms.csv')
dataset
```

Out[19]:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8119	e	k	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p
8120	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	n	o	p
8121	e	f	s	n	f	n	a	c	b	n	...	s	o	o	p	o	o	p
8122	p	k	y	n	f	y	f	c	n	b	...	k	w	w	p	w	o	e
8123	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p

8124 rows × 23 columns

In [20]:
```python
#Checking for null values:
dataset.isnull().sum()
```

Out[20]:
```
class                       0
cap-shape                   0
cap-surface                 0
cap-color                   0
bruises                     0
odor                        0
gill-attachment             0
gill-spacing                0
gill-size                   0
gill-color                  0
stalk-shape                 0
stalk-root                  0
stalk-surface-above-ring    0
stalk-surface-below-ring    0
stalk-color-above-ring      0
stalk-color-below-ring      0
veil-type                   0
veil-color                  0
ring-number                 0
ring-type                   0
spore-print-color           0
population                  0
habitat                     0
dtype: int64
```

In [21]:
```python
#Since the class column is the target, we will look at it's unique values:
dataset['class'].unique()
```

Out[21]: array(['p', 'e'], dtype=object)

This means we have a binary problem, classifying between p (poisonous) and e (edible).

In [22]:
```python
#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [23]:
```python
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[23]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatio
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [24]:
```python
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [25]:
```python
#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
                                                            test_size=0.19,
                                                            random_state=1)
print("X train_val: {}, X test: {}".format(X_train_val.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train_val.shape, y_test.shape))
```

```
X train_val: (6580, 117), X test: (1544, 117)
Y train_val: (6580,), Y test: (1544,)
```

In [26]:
```python
#Normalizing the features:
from sklearn.preprocessing import StandardScaler
normalizer = StandardScaler()
X_train_val = normalizer.fit_transform(X_train_val)
X_test = normalizer.transform(X_test)
```

In [27]:
```python
#Imports:
from random import shuffle
```

In [28]:
```python
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]


    #Converting back into Dataframes to be able to be used by my algorithm

    X_train_df = pd.DataFrame(X_train)
    y_train_df = pd.DataFrame(y_train, columns = ['class'])
    X_valid_df = pd.DataFrame(X_valid)
    y_valid_df = pd.DataFrame(y_valid, columns = ['class'])


    appended = pd.concat([X_train_df,y_train_df], axis = 1)
    appendedValid = pd.concat([X_valid_df,y_valid_df], axis = 1)

    treeShroom = id3(appended,'class')

    #Determine the accuracy score
    accuracy = evaluate(treeShroom, appendedValid, 'class') #evaluating the test dataset
    print("Model Accuracy for Fold ", (fold + 1), ": ", accuracy)

    sum_acc += accuracy

averageAcc = sum_acc/folds
print("Average Accuracy: {0:0.4f}".format(averageAcc))
```

```
[[3758 1579 1948 ... 4776 4110  334]
 [3212 5404 1129 ...  930 3552  235]
 [2288  305 2647 ... 4178 2502 3273]
 ...
 [1086   14 4687 ... 1001 5238 4782]
 [2003 1179 5716 ... 5589 4870 1249]
 [5508 4573   89 ... 3304  135 5348]]
Model Accuracy for Fold  1 :  1.0
Model Accuracy for Fold  2 :  1.0
Model Accuracy for Fold  3 :  1.0
Model Accuracy for Fold  4 :  1.0
Model Accuracy for Fold  5 :  1.0
Model Accuracy for Fold  6 :  1.0
Model Accuracy for Fold  7 :  1.0
Model Accuracy for Fold  8 :  1.0
Model Accuracy for Fold  9 :  0.9984802431610942
Model Accuracy for Fold  10 :  1.0
Average Accuracy: 0.9998
```

From this, we see that our model did extremely well--an accuracy of .9998--for the mushroom dataset. Only one of the folds did not output a perfect accuracy of 1, but that fold still had an extremely high accuracy. Thus, our model did well on this dataset. I will now compare this result with what the result would have been if I had used sklearn's Decision Tree Classifier:

In [29]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.metrics import accuracy_score
```

In [30]:
```python
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]

    clf_entropy = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)

    #Finally, fitting the model:
    clf_entropy.fit(X_train, y_train)

    y_pred_entropy = clf_entropy.predict(X_valid)
    y_pred_train_entropy = clf_entropy.predict(X_train) #for model comparison

    #Determine the accuracy score
    acc = accuracy_score(y_valid, y_pred_entropy)
    accTrain = accuracy_score(y_train, y_pred_train_entropy)

    print('Model accuracy score with criterion Entropy index: {0:0.4f} for fold'. format(acc), (fold + 1))
    #Accuracy Score for training set
    print('Training-set accuracy score: {0:0.4f}'. format(accTrain))

    sum_acc += acc

averageAcc = sum_acc/folds
print("Average Accuracy: {0:0.4f}".format(averageAcc))
```

```
[[ 421 4011 2530 ... 1292 6043 6095]
 [ 484 3375 1195 ... 1647 5962 3269]
 [5440  565 3523 ... 3866 2702 2384]
 ...
 [1300 5988 3093 ... 5167  910 2773]
 [2315 5550 6176 ... 1645 1714 6378]
 [1740  402 4248 ... 4492 5107  148]]
Model accuracy score with criterion Entropy index: 0.9620 for fold 1
Training-set accuracy score: 0.9612
Model accuracy score with criterion Entropy index: 0.9696 for fold 2
Training-set accuracy score: 0.9603
Model accuracy score with criterion Entropy index: 0.9590 for fold 3
Training-set accuracy score: 0.9615
Model accuracy score with criterion Entropy index: 0.9559 for fold 4
Training-set accuracy score: 0.9618
Model accuracy score with criterion Entropy index: 0.9681 for fold 5
Training-set accuracy score: 0.9605
Model accuracy score with criterion Entropy index: 0.9498 for fold 6
Training-set accuracy score: 0.9625
Model accuracy score with criterion Entropy index: 0.9590 for fold 7
Training-set accuracy score: 0.9615
Model accuracy score with criterion Entropy index: 0.9605 for fold 8
Training-set accuracy score: 0.9613
Model accuracy score with criterion Entropy index: 0.9650 for fold 9
Training-set accuracy score: 0.9608
Model accuracy score with criterion Entropy index: 0.9635 for fold 10
Training-set accuracy score: 0.9610
Average Accuracy: 0.9612
```

This suggests that our own model did better than the DecisionTreeClassifierModel (since the average accuracy score is a bit lower).

c) Compare test set accuracy when training a mushroom classifier using 500 vs. 5,000 training examples.

I will first find the test set accuracy when training a mushroom classifier using 500 training examples:

In [31]:
```python
#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [32]:
```python
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[32]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatio
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [33]:
```python
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [34]:
```python
#We will start by finding the test set accuracy when training a
#mushroom classifier using 500 training samples:

#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
                                                            test_size=0.9384,
                                                            random_state=1)
print("X train_val: {}, X test: {}".format(X_train_val.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train_val.shape, y_test.shape))
```

```
X train_val: (500, 117), X test: (7624, 117)
Y train_val: (500,), Y test: (7624,)
```

In [35]:
```python
#Normalizing the features:
from sklearn.preprocessing import StandardScaler
normalizer = StandardScaler()
X_train_val = normalizer.fit_transform(X_train_val)
X_test = normalizer.transform(X_test)
```

In [36]:
```python
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]


    #Converting back into Dataframes to be able to be used by my algorithm

    X_train_df = pd.DataFrame(X_train)
    y_train_df = pd.DataFrame(y_train, columns = ['class'])
    X_valid_df = pd.DataFrame(X_valid)
    y_valid_df = pd.DataFrame(y_valid, columns = ['class'])


    appended = pd.concat([X_train_df,y_train_df], axis = 1)
    appendedValid = pd.concat([X_valid_df,y_valid_df], axis = 1)

    treeShroom = id3(appended,'class')

    #Determine the accuracy score
    accuracy = evaluate(treeShroom, appendedValid, 'class') #evaluating the test dataset
    print("Model Accuracy for Fold ", (fold + 1), ": ", accuracy)

    sum_acc += accuracy

averageAcc = sum_acc/folds
print("Average Accuracy: {0:0.4f}".format(averageAcc))
```

```
[[179 254 380 387  13 296 316  15 382 143 347 445 216  65 337 144 345  50
  111 416 166 120 226 368 240 374 153 176 189 411 237 131 245  82 391 433
  364 169  47 152 469  78 308 409 174  90 466 342 132  28]
 [183 238 212  18 182 239 392  69 464 432 173  40 339 298 123 147 109  59
  210 295 310 442 354 202   1 289 252 162 475   5 246  76  51 359 395 446
  284 196  84 247  23 320 255 114 242 309 172 294 428 220]
 [344  12  46 201 461 130 103 251 232  31 139 355 300 180 393 318  43 317
  224 375 262 287 135 480 429  79  72 358  57  96 343 460 116 335 249 270
   21  86 315 151  95 398 443 236 360 269 348 108 186 419]
 [325 273 127  36 253 498 323 293 405 481  81 330  24 297 307 423 204 356
  324  55 438 363  38  74 372 303  87 149 286 197 399  77 319 155 340 396
  415 258  73 427 373 494 241   0  54 259 322 417 471 385]
 [379 256  62 388 349 458  29 486  66 161 133  11 205 277 203 218 207 488
  225 142  39  16 248 410 280 115 366 290 353 171   7  22 357 119 168 141
  462 302 276 121 126 292  67 455 448 478 321  20 158  75]
 [336 497  53 164 195 304 459 229 483 491 346 137 209 235 435 492 177 191
  400   6 412 275 181 402 150  34  26 447 431 499 414 313 122   8 408 268
  365 384 473 441 217  63 436  58 468 188 361 386 272 193]
 [332 107 211 154 175 208 184 378 456   3 381 426 223  56 125  83 160  10
  314 306 138 489 450 333  32 278   2 394 477  17 404 190  64 200 457 104
  370  88 312 101  85 136 467 329 215 437 260 377  71  19]
 [185 487 453 451 106 465 234 406  94 146 129 113 227 401 165 403 424   4
   70 291 199 124 413 230 283  61 496 156  25 117 367 159  41 422 244 362
  485 479 194 167 482  60 145 495 299 338  99 267  30 407]
 [187  98 157 454 266 257 219 301  42 221 420 250  45  52  97 100 233 421
  327 389 463  44 105 452 328 192 390 474 274 484 279 261 434  93 140 326
  213 444 334 425 439 222 376 214  89 112  49  27 263 206]
 [ 92  37  14 134  48  80 305 331 440  33 198 470   9  35 351 271 490 383
  178 265 128 118  91 472 352 170 350 493 163 264 369 371 102 110 228 281
  341 476  68 449 311 285 430 397 243 148 282 418 288 231]]
Model Accuracy for Fold  1 :  1.0
Model Accuracy for Fold  2 :  1.0
Model Accuracy for Fold  3 :  1.0
Model Accuracy for Fold  4 :  0.96
Model Accuracy for Fold  5 :  1.0
Model Accuracy for Fold  6 :  0.98
Model Accuracy for Fold  7 :  1.0
Model Accuracy for Fold  8 :  1.0
Model Accuracy for Fold  9 :  1.0
Model Accuracy for Fold  10 :  1.0
Average Accuracy: 0.9940
```

Thus, we see the average accuracy is .9940 for 500 training samples. We will perform the same calculations for 5000 training samples:

In [37]:
```python
#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [38]:
```python
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[38]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatior
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [39]:

```
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [40]:

```
#We will start by finding the test set accuracy when training a
#mushroom classifier using 500 training samples:

#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
                                                            test_size=0.3845,
                                                            random_state=1)
print("X train_val: {}, X test: {}".format(X_train_val.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train_val.shape, y_test.shape))
```

```
X train_val: (5000, 117), X test: (3124, 117)
Y train_val: (5000,), Y test: (3124,)
```

In [41]:

```
#Normalizing the features:
from sklearn.preprocessing import StandardScaler
normalizer = StandardScaler()
X_train_val = normalizer.fit_transform(X_train_val)
X_test = normalizer.transform(X_test)
```

In [42]:

```
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]


    #Converting back into Dataframes to be able to be used by my algorithm

    X_train_df = pd.DataFrame(X_train)
    y_train_df = pd.DataFrame(y_train, columns = ['class'])
    X_valid_df = pd.DataFrame(X_valid)
    y_valid_df = pd.DataFrame(y_valid, columns = ['class'])


    appended = pd.concat([X_train_df,y_train_df], axis = 1)
    appendedValid = pd.concat([X_valid_df,y_valid_df], axis = 1)

    treeShroom = id3(appended,'class')

    #Determine the accuracy score
    accuracy = evaluate(treeShroom, appendedValid, 'class') #evaluating the test dataset
    print("Model Accuracy for Fold ", (fold + 1), ": ", accuracy)

    sum_acc += accuracy

averageAcc = sum_acc/folds
print("Average Accuracy: {0:0.4f}".format(averageAcc))
```

```
[[1560 3203 1584 ... 3631 4474 3178]
 [3375  570 3908 ... 1415 2142  692]
 [  65 3217 2797 ... 3122 4579 3787]
 ...
 [4312 1351 4005 ... 2580  964  787]
 [ 115  191 4855 ... 3890 3004 4198]
 [ 594 1128 2013 ... 1169 4216 4015]]
Model Accuracy for Fold  1 :  1.0
Model Accuracy for Fold  2 :  1.0
Model Accuracy for Fold  3 :  1.0
Model Accuracy for Fold  4 :  1.0
Model Accuracy for Fold  5 :  1.0
Model Accuracy for Fold  6 :  1.0
Model Accuracy for Fold  7 :  1.0
Model Accuracy for Fold  8 :  1.0
Model Accuracy for Fold  9 :  0.998
Model Accuracy for Fold  10 :  1.0
Average Accuracy: 0.9998
```

Thus, when using our model on 5000 training samples, we achieve a test score accuracy of .9998. Since the test score accuracy for 500 training samples was .9940, we see that the larger training dataset allowed the model to perform slightly better, as expected.

d) Report accuracy on training and test data when using a decision tree that has 10 vs 30 leaves.

I am a bit confused how we are supposed to complete part d. It was not specified in part a that our model should account for/allow us to input a certain number of leaves. Thus, the model I created cannot do that. I am not going to rewrite my entire model to account for this--which I already had trouble doing since I was never taught how to do this--so I need to use the sklearn's Decision Tree Classifier. If our model should be able to account for this, it should have been specified in part a, not in part d when a student has already spent the time building the model a certain way. I do not feel that this is really my fault, so I hope you do not take off points for me using the Decision Tree Classifier.

In [43]:

```
#First, I will find the accuracy on the training and test data when using
# a decision tree that has 10 leaves:

#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [44]:

```
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[44]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatior
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [45]:

```
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [46]:

```
#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
                                                            test_size=0.3845,
                                                            random_state=1)
print("X train_val: {}, X test: {}".format(X_train_val.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train_val.shape, y_test.shape))
```

```
X train_val: (5000, 117), X test: (3124, 117)
Y train_val: (5000,), Y test: (3124,)
```

In [47]:

```
#Normalizing the features:
from sklearn.preprocessing import StandardScaler
normalizer = StandardScaler()
X_train_val = normalizer.fit_transform(X_train_val)
X_test = normalizer.transform(X_test)
```

In [48]:

```
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0
sum_train_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]

    #WE NOTE THAT WE HAVE A MAX LEAF NODES OF 10 FOR 10 LEAVES
    clf_entropy = DecisionTreeClassifier(criterion='entropy', max_leaf_nodes=10, random_state=0)

    #Finally, fitting the model:
    clf_entropy.fit(X_train, y_train)

    y_pred_entropy = clf_entropy.predict(X_valid)
    y_pred_train_entropy = clf_entropy.predict(X_train) #for model comparison

    #Determine the accuracy score
    acc = accuracy_score(y_valid, y_pred_entropy)
    accTrain = accuracy_score(y_train, y_pred_train_entropy)

    print('Model accuracy score with criterion Entropy index: {0:0.4f} for fold'. format(acc), (fold + 1))
    #Accuracy Score for training set
    print('Training-set accuracy score with criterion Entropy index: {0:0.4f}'. format(accTrain))

    sum_acc += acc
    sum_train_acc += accTrain

averageAcc = sum_acc/folds
averageAccTrain = sum_train_acc/folds

print("Average Accuracy Test: {0:0.4f}".format(averageAcc))
print("Average Accuracy Train: {0:0.4f}".format(averageAccTrain))
```

```
[[2400 1499 1116 ... 3837 3620 4292]
 [4232 2842 1865 ... 1968 2945  796]
 [ 958 3552 4487 ... 1206 3987  968]
 ...
 [2245 2405 1944 ... 3769 4577 3335]
 [4926 2457 2722 ... 2320  264 1571]
 [1283 1424 4276 ... 1754 2748 4904]]
Model accuracy score with criterion Entropy index: 1.0000 for fold 1
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 2
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 3
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 4
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 5
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 6
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 7
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 1.0000 for fold 8
Training-set accuracy score with criterion Entropy index: 0.9993
Model accuracy score with criterion Entropy index: 0.9960 for fold 9
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 0.9980 for fold 10
Training-set accuracy score with criterion Entropy index: 0.9998
Average Accuracy Test: 0.9994
Average Accuracy Train: 0.9994
```

Thus, the average accuracy on the training data is .9994 and on the testing data is .9994 when using 10 leaves.

We will now do the same procedure but with 30 leaves:

In [49]:

```
#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [50]:

```
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[50]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatior
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [51]:

```
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [52]:

```
#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y,
                                                            test_size=0.3845,
                                                            random_state=1)
print("X train_val: {}, X test: {}".format(X_train_val.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train_val.shape, y_test.shape))
```

```
X train_val: (5000, 117), X test: (3124, 117)
Y train_val: (5000,), Y test: (3124,)
```

In [53]:

```
#Normalizing the features:
from sklearn.preprocessing import StandardScaler
normalizer = StandardScaler()
X_train_val = normalizer.fit_transform(X_train_val)
X_test = normalizer.transform(X_test)
```

In [54]:

```
#We would like to use 10-fold cross validation:
folds = 10

#Getting the number of samples in the training and validation set
num_train_val = X_train_val.shape[0]

# shuffle the index of samples in the train_val set
index_of_samples = np.arange(num_train_val)
shuffle(index_of_samples)

# split the index of the train_valid set into 10 folds
index_of_folds = index_of_samples.reshape(folds, -1)
print(index_of_folds)

best_acc = 0.0

#10-fold cross-validation:
sum_acc = 0.0
sum_train_acc = 0.0

for fold in range(folds):
    index_of_folds_temp = index_of_folds.copy()

    valid_index = index_of_folds_temp[fold,:].reshape(-1) #get the index of the validation set
    train_index = np.delete(index_of_folds_temp, fold, 0).reshape(-1) #get the index of the training set

    # training set
    X_train = X_train_val[train_index]
    y_train = y_train_val[train_index]

    # validation set
    X_valid = X_train_val[valid_index]
    y_valid = y_train_val[valid_index]

    #WE NOTE THAT WE HAVE A MAX LEAF NODES OF 30 FOR 30 LEAVES
    clf_entropy = DecisionTreeClassifier(criterion='entropy', max_leaf_nodes=30, random_state=0)

    #Finally, fitting the model:
    clf_entropy.fit(X_train, y_train)

    y_pred_entropy = clf_entropy.predict(X_valid)
    y_pred_train_entropy = clf_entropy.predict(X_train) #for model comparison

    #Determine the accuracy score
    acc = accuracy_score(y_valid, y_pred_entropy)
    accTrain = accuracy_score(y_train, y_pred_train_entropy)

    print('Model accuracy score with criterion Entropy index: {0:0.4f} for fold'. format(acc), (fold + 1))
    #Accuracy Score for training set
    print('Training-set accuracy score with criterion Entropy index: {0:0.4f}'. format(accTrain))

    sum_acc += acc
    sum_train_acc += accTrain

averageAcc = sum_acc/folds
averageAccTrain = sum_train_acc/folds

print("Average Accuracy Test: {0:0.4f}".format(averageAcc))
print("Average Accuracy Train: {0:0.4f}".format(averageAccTrain))
```

```
[[2754  657 2527 ... 3290  109 1583]
 [ 116 1243 4340 ... 1982 3417 3480]
 [ 932  711 2103 ... 3568 1660 3166]
 ...
 [ 295 3917  793 ... 1740 1601  197]
 [3886 3207 3710 ... 1422 2446 4010]
 [4759 4070 2615 ...  268   13 2988]]
Model accuracy score with criterion Entropy index: 0.9980 for fold 1
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 2
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 3
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 4
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 5
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 6
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 7
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 8
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 9
Training-set accuracy score with criterion Entropy index: 1.0000
Model accuracy score with criterion Entropy index: 1.0000 for fold 10
Training-set accuracy score with criterion Entropy index: 1.0000
Average Accuracy Test: 0.9998
Average Accuracy Train: 1.0000
```

With 30 leaves, we see the average accuracy for the testing data is .998 and 1.0 for the training data. These scores are only slightly better than with 10 leaves, suggesting that having a max depth of 10 (having 10 leaves) was sufficient enough. Still, a max_leaf_node of 30 seems to improve the accuracy ever so slightly, so we most likely would want just over 10 leaves.

e) Report attribute tests used in the decision tree with 10 leaves.

I am a bit confused what this question is asking. I have looked up the term "attribute tests" and have found no results. This terminology appears to be unique to the person who wrote this assignment, as I cannot find anything on it. The closest thing I could find online is "test attributes", but this does not make sense in this context. If I were to guess, I would say that this question is asking us to discuss how the accuracy score was determined.

The accuracy score outputted was determined using sklearn's accuracy_score function, which takes as parameters the ground truth (correct) labels, and the predicted labels. If these values match exactly, then there will be an accuracy score of 1.0. In the decision tree model written above, this accuracy score function is run on the training data and the validation data to meet the desires of the previous question. The scores get printed for each fold, with their average taken at the end. The scores use criterion Entropy index, as specified when making the Decision Tree Classifier

f) For a decision tree with 10 leaves report the number of positive and negative training examples at each internal node and at leaves.

To report the number of positive and negative training examples at each internal node and at leaves, one would have to examine these numbers 10 times since we put the model on 10 folds. Since this would be a lot of information to examine, I will report these numbers without doing the 10-fold cross validation.

In [55]:

```
#Separating Features and Target:
X = dataset.drop(['class'], axis = 1)
y = dataset['class']
```

In [56]:

```
#Since the values in the dataset are all categorical, we can encode
#using pandas dummy variable and encode y using LabelEncoder.
X = pd.get_dummies(X)
X.head()
```

Out[56]:

	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s	cap-surface_y	...	population_s	population_v	populatior
0	0	0	0	0	0	1	0	0	1	0	...	1	0	
1	0	0	0	0	0	1	0	0	1	0	...	0	0	
2	1	0	0	0	0	0	0	0	1	0	...	0	0	
3	0	0	0	0	0	1	0	0	0	1	...	1	0	
4	0	0	0	0	0	1	0	0	1	0	...	0	0	

5 rows × 117 columns

In [57]:

```
#Encoding the target
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
y = encoder.fit_transform(y)
print(y)
```

```
[1 0 0 ... 0 1 0]
```

In [58]:

```
#Splitting dataset to training and test and printing the sizes:
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3845,
                                                    random_state=1)
print("X train_val: {}, X test: {}".format(X_train.shape, X_test.shape))
print("Y train_val: {}, Y test: {}".format(y_train.shape, y_test.shape))
```

```
X train_val: (5000, 117), X test: (3124, 117)
Y train_val: (5000,), Y test: (3124,)
```

In [59]:

```
#WE NOTE THAT WE HAVE A MAX LEAF NODES OF 10 FOR 10 LEAVES
clf_entropy = DecisionTreeClassifier(criterion='entropy', max_leaf_nodes=10, random_state=0)

#Finally, fitting the model:
clf_entropy.fit(X_train, y_train)

y_pred_entropy = clf_entropy.predict(X_test)
y_pred_train_entropy = clf_entropy.predict(X_train) #for model comparison

#Determine the accuracy score
acc = accuracy_score(y_test, y_pred_entropy)
accTrain = accuracy_score(y_train, y_pred_train_entropy)

print('Model accuracy score with criterion Entropy index: {0:0.4f}'. format(acc))
#Accuracy Score for training set
print('Training-set accuracy score with criterion Entropy index: {0:0.4f}'. format(accTrain))
```

```
Model accuracy score with criterion Entropy index: 0.9984
Training-set accuracy score with criterion Entropy index: 0.9994
```

In [60]:

```
import matplotlib.pyplot as plt
from sklearn import tree
```

In [61]:

```
plt.figure(figsize = (12,8))
tree.plot_tree(clf_entropy)
```

Out[61]:

```
[Text(386.3076923076923, 398.64, 'X[27] <= 0.5\nentropy = 0.999\nsamples = 5000\nvalue = [2600, 2400]'),
 Text(257.53846153846155, 326.15999999999997, 'X[20] <= 0.5\nentropy = 0.663\nsamples = 2818\nvalue = [485, 233
3]'),
 Text(206.03076923076924, 253.67999999999998, 'X[53] <= 0.5\nentropy = 0.978\nsamples = 825\nvalue = [485, 34
0]'),
 Text(154.52307692307693, 181.2, 'X[63] <= 0.5\nentropy = 0.934\nsamples = 523\nvalue = [183, 340]'),
 Text(103.01538461538462, 108.71999999999997, 'X[110] <= 0.5\nentropy = 0.645\nsamples = 407\nvalue = [67, 34
0]'),
 Text(51.50769230769231, 36.23999999999995, 'entropy = 0.0\nsamples = 340\nvalue = [0, 340]'),
 Text(154.52307692307693, 36.23999999999995, 'entropy = 0.0\nsamples = 67\nvalue = [67, 0]'),
 Text(206.03076923076924, 108.71999999999997, 'entropy = 0.0\nsamples = 116\nvalue = [116, 0]'),
 Text(257.53846153846155, 181.2, 'entropy = 0.0\nsamples = 302\nvalue = [302, 0]'),
 Text(309.04615384615386, 253.67999999999998, 'entropy = 0.0\nsamples = 1993\nvalue = [0, 1993]'),
 Text(515.0769230769231, 326.15999999999997, 'X[100] <= 0.5\nentropy = 0.198\nsamples = 2182\nvalue = [2115, 6
7]'),
 Text(463.5692307692308, 253.67999999999998, 'X[63] <= 0.5\nentropy = 0.086\nsamples = 2138\nvalue = [2115, 2
3]'),
 Text(360.55384615384617, 181.2, 'X[35] <= 0.5\nentropy = 0.016\nsamples = 2109\nvalue = [2106, 3]'),
 Text(309.04615384615386, 108.71999999999997, 'entropy = 0.164\nsamples = 124\nvalue = [121, 3]'),
 Text(412.0615384615385, 108.71999999999997, 'entropy = 0.0\nsamples = 1985\nvalue = [1985, 0]'),
 Text(566.5846153846154, 181.2, 'X[89] <= 0.5\nentropy = 0.894\nsamples = 29\nvalue = [9, 20]'),
 Text(515.0769230769231, 108.71999999999997, 'entropy = 0.0\nsamples = 20\nvalue = [0, 20]'),
 Text(618.0923076923077, 108.71999999999997, 'entropy = 0.0\nsamples = 9\nvalue = [9, 0]'),
 Text(566.5846153846154, 253.67999999999998, 'entropy = 0.0\nsamples = 44\nvalue = [0, 44]')]
```

We now can see the number of positive and negative training examples at each internal node and at leaves for a decision tree with 10 leaves. We note that the column names are not the same as the original dataset due to the label encode and dummy encoder we decided to use on the dataset. Nonetheless, we see that the tree has 10 leaves (as expected) with 6 levels and the number of positive and negative training examples reported in the 'value = []' section of each node. For the most part, values are reported as either positive or negative (indicated in the value brackets by one of the values being 0 and the other being the total of the rest of the values given for the parent node), which is emphasized by our model's high accuracy score. Since there are 10 leaf nodes and 9 internal nodes, I will not explicitly write out the positive and negative training examples for each one (especially since we can see those values easily in the tree pictured above). However, for an example, we see that the very first internal node, the root, has a split of 2600 and 2400.

Problem 2:

Show that the entropy of a node in a decision tree never increases after splitting it into smaller successor nodes.

The professor has noted that we do not need to complete this question if we are undergraduate students. Even though I am an undergraduate student, I will try to complete this proof.

Due to the large number of mathematical symbols required for this proof, I have written it in latex. However, I do not know how to paste Latex into a Jupyter Notebook, so I am putting pictures of my work/steps below:

Let $Y = \{y_1, y_2, ..., y_c\}$ where c represents the number of classes in attribute Y (the attribute we are hoping to predict). Also, let $X = \{x_1, x_2, \ldots, x_k\}$ where k is the number of attribute values a certain attribute X has. In the very beginning, when we are at the root of the tree, the entropy is given by the following equation:

$$E(Y) = -\Sigma_{j=1}^{c} P(y_j) log_2 P(y_j) = \Sigma_{j=1}^{c} \Sigma_{i=1}^{k} P(x_i, y_j) log_2 P(y_j),$$

which uses the following idea from the law of total probability: $P(y_j) = \Sigma_{i=1}^{k} P(x_i, y_j)$. After we split an attribute X, the entropy of each child node $X = x_i$ is

$$E(Y \mid x_i) = -\Sigma_{j=1}^{c} P(y_j \mid x_i) log_2 P(y_j \mid x_i),$$

where the proportion of the sample where $X = x_i$ belong to class y_j is represented by $P(y_j|x_i)$. Then, after we split X, the weighted entropy of the subsequent, child nodes is as follows:

$$\begin{aligned} E(Y \mid X) &= \Sigma_{i=1}^{k} P(x_i) E(Y \mid x_i) \\ &= -\Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(x_i) P(y_j \mid x_i) log_2 P(y_j \mid x_i) \\ &= -\Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) log_2 P(y_j \mid x_i), \end{aligned}$$

from the law of total probability. To prove that the entropy never increases after splitting a node, we must show that $E(Y \mid X) \leq E(Y \mid X)$, which we can do by showing that $E(Y \mid X) - E(Y \mid X) \leq 0$.

$$\begin{aligned} E(Y \mid X) - E(Y) &= -\Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) log_2 P(y_j \mid x_i) + \Sigma_{j=1}^{c} P(y_j, x_i) log_2 P(y_j \mid x_i) \\ &= \Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) log_2 P(y_j \mid x_i) - P(y_j, x_i) log_2 P(y_j) \\ &= \Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) log_2 \frac{P(y_j)}{P(y_j \mid x_i)} \\ &= \Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) log_2 \frac{P(y_j) P(x_i)}{P(y_j, x_i)} \\ &\leq log_2 \left(\Sigma_{i=1}^{k} \Sigma_{j=1}^{c} P(y_j, x_i) \frac{P(y_j) P(x_i)}{P(y_j, x_i)}\right) : by\ Jensen's\ Inequality \\ &= log_2 [\Sigma_{i=1}^{k} P(x_k) \Sigma_{j=1}^{c} P(y_j)] \\ &= log_2(1) \\ &= 0 \end{aligned}$$

Thus, the entropy of a node in a decision tree never increases after splitting it into smaller successor nodes. //

Problem 3:

Consider two classifiers A and B. On one data set, 10-fold cross-validation shows that classifier A is better than B by 3%, with a standard deviation of 7% over 100 different folds. On the other data set, classifier B is better than classifier A by 1%, with a standard deviation of 0.1% over 100 different folds. Which classifier would you prefer on the basis of this evidence, and why?

Based on the given information, I would prefer classifier B over classifier A. While classifier A's improvement is 2% improvement, on average, greater than classifier B's improvement, classifier A's distribution is much wider. This wider distribution suggests that there is less precision and reliability each time. Additionally, since the accuracy of classifier B is less than 1 standard deviation away from the accuracy of classifier A, classifier B is not giving up much to reduce variance. This makes classifier B more preferable, in my opinion.

Problem 4:

Discuss the advantages and disadvantages of the nearest neighbor classifier, over a decision tree.

There are numerous advantages and disadvantages of the nearest neighbor classifier and decision tree. Both the nearest neighbor (KNN) and the decision tree (DT) methods are non-parametric, suggesting that the inference does not require an underlying distribution of the data to be assumed. KNN can produce boundaries of arbitrary shapes; on the other hand, DTs can only produce rigid, right-angled boundaries (rectangular shaped). At the same time, a KNN classifier is sensitive to missing values and noise and requires all of the data to be numeric and scaled to prevent one attribute from dominating the others. In contrast, DTs are robust to noise, are inexpensive to construct, and allow for non-numeric data. Still, decision trees are flawed because space is exponentially large, greedy approaches often fail to find the best tree, decision trees do not handle interactions, and each boundary can only involve a single attribute at a time. Thus, each classifier has its advantages and disadvantages.

Decision Trees:

Advantages:

- Require less effort for data preparation during pre-processing than other algorithms
- Does not need normalization of data
- Does not need scaling of data
- Missing values do NOT affect the process of building a decision tree considerably
- Very intuitive and easy to explain to technical teams as well as stakeholders

Disadvantages:

- A small change in the data can cause a large change in the structure of the decision tree
- Calculations can be more eccentric than other algorithms
- May need more time to train
- Training is relatively expensive
- Inadequate for applying regression and predicting continuous values

Nearest Neighbor:

Advantages:

- No training period, which makes the algorithm faster
- New data can be added seamlessly
- Easy to implement
- Can be used for both classification and regression
- Variety of distance criteria to choose from

Disadvantages:

- Does not work well on larger datasets (slow)
- Does not work well on higher dimension datasets (curse of dimensionality)
- Needs feature scaling (standardization and normalization)
- Sensitive to noisy data, missing values, and outliers
- Need to figure out the optimal number of neighbors