Problem 3: Poisonous Mushrooms?

In [1]:
```
import numpy as np
import pandas as pd
import math
from collections import deque
```

In [2]:
```
labels = 'class, cap-shape, cap-surface, cap-color, bruises, odor, gill-attach
ment, gill-spacing, gill-size, '\
'gill-color, stalk-shape, stalk-root,' \
'stalk-surface-above-ring, stalk-surface-below-ring, stalk-color-above-ring, s
talk-color-below-ring, veil-type, veil-color,'\
'ring-number, ring-type, spore-print-color, population, habitat'.split(',')
labels = [label.strip() for label in labels]
label_idx = {label:idx for idx, label in enumerate(labels)}
```

In [3]:
```
df_train = pd.read_csv('mush_train.data',header=None, names=labels)
df_test = pd.read_csv('mush_test.data', header=None, names=labels)
```

In [4]:
```
y_train, X_train = df_train.iloc[:, 0], df_train.iloc[:, 1:]
y_test, X_test = df_test.iloc[:, 0], df_test.iloc[:, 1:]
```

In [5]:
```
class InternalNode:

    def __init__(self, attr, attr_vals, data, ig=None, height=0):
        self.attr = attr
        self.children = {attr_val:None for attr_val in attr_vals}
        self.children_count = 0
        self.ig = ig
        self.height = height
        self.data = data

    def set_child(self, attr_val, child):
        self.children[attr_val] = child
        self.children_count += 1

    def get_children(self):
        return self.children.items()

    def get_children_count(self):
        return self.children_count

    def __repr__(self):
        return f"Attribute: {self.attr}({label_idx[self.attr]}) Attrs: {self.c
hildren_count} IG: {self.ig}"

class LeafNode:

    def __init__(self, attr, attr_val, prediction, height):
        self.attr = attr
        self.attr_val = attr_val
        self.prediction = prediction
        self.height = height

    def predict(self):
        return self.prediction


    def __repr__(self):
        return f"Attribute: {self.attr}({label_idx[self.attr]}), Prediction:
{self.prediction}"
```

In [6]:

```
def find_best_split(data, attributes):
    # Finds best attribute to split on based on Conditional entropy (IG)
    m, n = data.shape
    min_entropy = 1
    best_attr = -1
    for attr in attributes:
        cond_entropy = 0
        attr_vals, counts = np.unique(data[attr], return_counts=True)
        for attr_val, attr_count in zip(attr_vals, counts):
            sub_data = data[data[attr] == attr_val]
            subset_len, _ = sub_data.shape
            subclass_counts = sub_data['class'].value_counts()
            p = subclass_counts['p'] if 'p' in subclass_counts else 0
            e = subclass_counts['e'] if 'e' in subclass_counts else 0
            plogp, eloge = 0, 0
            if p > 0:
                plogp = - (p/subset_len) * math.log2(p/subset_len)
            if e > 0:
                eloge = - (e/subset_len) * math.log2(e/subset_len)
            cond_entropy += (attr_count/m) * plogp * eloge
        if cond_entropy <= min_entropy:
            if cond_entropy == min_entropy and label_idx[attr] > label_idx[bes
t_attr]:
                # In case of a tie, first occurring attribute is used
                continue
            min_entropy = cond_entropy
            best_attr = attr
    return (best_attr, entropy - min_entropy)
```

In [7]:
```
def fit(data, attributes, root=None):
    split_attr, ig = find_best_split(data, attributes)
    attributes.remove(split_attr)
    if not root:
        root = InternalNode(split_attr, data[split_attr].unique(), data, ig)
    # Using queue to construct the tree
    queue = deque()
    queue.append(root)
    while queue:
        current_node = queue.popleft()
        current_node_attr = current_node.attr

        # Data filtered with current attribute value
        current_data = current_node.data
        for attr_val, child in current_node.get_children():
            new_node = None

            # Create new dataset with attribute = attribute value
            subset_data = current_data[current_data[current_node_attr] == attr
_val]
            subset_len, _ = subset_data.shape
            subclass_counts = subset_data['class'].value_counts()
            p = subclass_counts['p'] if 'p' in subclass_counts else 0
            e = subclass_counts['e'] if 'e' in subclass_counts else 0
            if p == subset_len:
                new_node = LeafNode(current_node_attr, attr_val, 'p', current_
node.height + 1)
            elif e == subset_len:
                new_node = LeafNode(current_node_attr, attr_val, 'e', current_
node.height + 1)
            else:
                split_attr, ig = find_best_split(subset_data, attributes)
                attributes.remove(split_attr)
                new_node = InternalNode(split_attr, data[split_attr].unique(),
subset_data, ig, current_node.height + 1)
                queue.append(new_node)
            current_node.set_child(attr_val, new_node)
    return root
```

In [8]:
```
m, n = df_train.shape
global_class_counts = df_train['class'].value_counts()
p_p = global_class_counts['p']/m
p_e = global_class_counts['e']/m
entropy = - (p_p * math.log2(p_p)) - (p_e * math.log2(p_e))
```

In [9]:
```
root = fit(df_train, X_train.columns.to_list())
```

In [10]:
```
dummy_print = lambda x: x
def print_tree(root, print):
    q = deque()
    q.append(root)
    print(root)
    max_height = 0
    while q:
        e = q.popleft()
        for key, val in e.get_children():
            if isinstance(val, LeafNode):
                print('\t'*val.height + f'{key} -> {val.prediction}')
            elif val is not None:
                print('\t'*val.height + f'{key} -> {val}')
                q.append(val)
            if val is not None and val.height > max_height:
                max_height = val.height
    return max_height
```

1. Assuming you break ties using the attribute that occurs first (left to right) in the data, draw the resulting decision tree and report the maximum information gain for each node that you added to the tree.

In [11]:
```
print("Depth = ", print_tree(root, print=dummy_print))
```

```
Depth =  4
```

In [12]:
```
def predict(test, root):
    m, n = test.shape

    def tree_predictor(row):
        predicted = False
        current = root
        while not predicted:
            row_val = row[current.attr]
            next_node = current.children[row_val]
            if isinstance(next_node, LeafNode):
                return next_node.predict()
            else:
                current = next_node
        return None

    return test.apply(tree_predictor, axis=1)

def get_accuracy(y_pred, y_test):
    return 100 * np.mean(y_pred.ravel() == y_test.ravel())
```

2. What is the accuracy of this decision tree on the test data?

In [13]:
```
y_pred_test = predict(X_test, root)
print("Testing Accuracy:",  get_accuracy(y_pred_test, y_test))
```

```
Testing Accuracy: 100.0
```

3. Now consider arbitrary input data. Suppose that you decide to limit yourself to decision trees of height one, i.e., only one split. Is the tree produced by the information gain heuristic optimal on the training data (that is, no other decision tree has higher accuracy)?

IG Heuristic is a greedy algorithm and it provides a good approxmiation for deciding on best attribute to split on. It need not be the case that it always creates the most optimal decision tree, as it only looks for local optimal value rather than global optimal split. However, **for one trees of height one, it provides the most optimal decision tree.** as the local optimum is the global optimum

In [14]:
```
def split_all(data, attributes):
    # Generate all decision trees with height = 1
    m, n = data.shape
    split_nodes = []
    for attr in attributes:
        cond_entropy = 0
        attr_vals, counts = np.unique(data[attr], return_counts=True)
        root = InternalNode(attr, attr_vals, data)
        for attr_val, attr_count in zip(attr_vals, counts):
            sub_data = data[data[attr] == attr_val]
            subset_len, _ = sub_data.shape
            subclass_counts = sub_data['class'].value_counts()
            p = subclass_counts['p'] if 'p' in subclass_counts else 0
            e = subclass_counts['e'] if 'e' in subclass_counts else 0
            plogp, eloge = 0, 0
            if p > 0:
                plogp = - (p/subset_len) * math.log2(p/subset_len)
            if e > 0:
                eloge = - (e/subset_len) * math.log2(e/subset_len)
            child_node = LeafNode(attr, attr_val, 'e' if p < e else 'p', root.
height+1)
            root.set_child(attr_val, child_node)
            cond_entropy += (attr_count/m) * plogp * eloge
        root.ig = entropy-cond_entropy
        split_nodes.append(root)
    return split_nodes
```

In [17]:
```
level_1_nodes = split_all(df_train, X_train.columns)
```

In [27]:
```
import operator
level_1_nodes.sort(key=operator.attrgetter('ig'))
level_1_nodes.reverse()
```

In [30]:
```
df = {
    'Level 1 Split': [],
    'Information Gain': [],
    'Train data Accuracy': []
}
for node in level_1_nodes:
    df['Level 1 Split'].append(node.attr)
    df['Information Gain'].append(node.ig)
    df['Train data Accuracy'].append(get_accuracy(predict(X_train, node), y_tr
ain))

pd.DataFrame.from_dict(df)
```

Out[30]:

	Level 1 Split	Information Gain	Train data Accuracy
0	odor	0.995875	98.556876
1	spore-print-color	0.929495	87.202886
2	gill-color	0.888091	80.751273
3	ring-type	0.861894	77.313243
4	stalk-surface-above-ring	0.861587	77.228353
5	stalk-surface-below-ring	0.857274	76.464346
6	gill-size	0.854195	76.018676
7	bruises	0.841670	74.320883
8	stalk-color-above-ring	0.828693	71.561969
9	population	0.824511	71.625637
10	stalk-color-below-ring	0.824095	70.628183
11	habitat	0.816664	69.057725
12	stalk-root	0.808559	65.343803
13	gill-spacing	0.788198	61.608659
14	cap-shape	0.771600	56.706282
15	cap-color	0.767437	59.337861
16	cap-surface	0.765277	58.043294
17	ring-number	0.763740	53.650255
18	veil-color	0.755415	51.782683
19	gill-attachment	0.754834	51.655348
20	stalk-shape	0.753921	55.305603
21	veil-type	0.749658	51.655348

Observing the results from the above table, Training data accuracy increases with respect to Information Gain (IG) with trees height restricted to one and hence IG provides a good measure for splitting attribute.