```
import numpy as np
import matplotlib.pyplot as plt
from public_tests import *

%matplotlib inline


X_train = np.array([[1,1,1],[1,0,1],[1,0,0],[1,0,0],[1,1,1],[0,1,1],[0,0,0],[1,0,1],[0,1,0],[1,0,0]])
y_train = np.array([1,1,0,0,1,0,0,1,1,0])


print("First few elements of X_train:\n", X_train[:5])
print("Type of X_train:",type(X_train))
```

```
First few elements of X_train:
 [[1 1 1]
 [1 0 1]
 [1 0 0]
 [1 0 0]
 [1 1 1]]
Type of X_train: <class 'numpy.ndarray'>
```

```
print("First few elements of y_train:", y_train[:5])
print("Type of y_train:",type(y_train))
```

```
First few elements of y_train: [1 1 0 0 1]
Type of y_train: <class 'numpy.ndarray'>
```

```
print ('The shape of X_train is:', X_train.shape)
print ('The shape of y_train is: ', y_train.shape)
print ('Number of training examples (m):', len(X_train))
```

```
The shape of X_train is: (10, 3)
The shape of y_train is:  (10,)
Number of training examples (m): 10
```

```
def compute_entropy(y):
    """
    Computes the entropy for

    Args:
       y (ndarray): Numpy array indicating whether each example at a node is
           edible (`1`) or poisonous (`0`)

    Returns:
        entropy (float): Entropy at that node

    """
    # You need to return the following variables correctly
    entropy = 0.

    ### START CODE HERE ###
    if len(y) != 0:
        p1 = p1 = len(y[y == 1]) / len(y)
     # For p1 = 0 and 1, set the entropy to 0 (to handle 0log0)
        if p1 != 0 and p1 != 1:
             entropy = -p1 * np.log2(p1) - (1 - p1) * np.log2(1 - p1)
        else:
             entropy = 0
    ### END CODE HERE ###

    return entropy


print("Entropy at root node: ", compute_entropy(y_train))

# UNIT TESTS
compute_entropy_test(compute_entropy)
```

```
Entropy at root node:  1.0
 All tests passed.
```

```
def split_dataset(X, node_indices, feature):
    """
    Splits the data at the given node into
    left and right branches

    Args:
        X (ndarray):             Data matrix of shape(n_samples, n_features)
        node_indices (list):  List containing the active indices. I.e, the samples being considered at this step.
        feature (int):           Index of feature to split on

    Returns:
        left_indices (list): Indices with feature value == 1
        right_indices (list): Indices with feature value == 0
    """

    # You need to return the following variables correctly
    left_indices = []
    right_indices = []

    ### START CODE HERE ###
    for i in node_indices:
        if X[i][feature] == 1:
            left_indices.append(i)
        else:
            right_indices.append(i)
    ### END CODE HERE ###

    return left_indices, right_indices


root_indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Feel free to play around with these variables
# The dataset only has three features, so this value can be 0 (Brown Cap), 1 (Tapering Stalk Shape) or 2 (Solitary)
feature = 0

left_indices, right_indices = split_dataset(X_train, root_indices, feature)

print("Left indices: ", left_indices)
print("Right indices: ", right_indices)

# UNIT TESTS
split_dataset_test(split_dataset)
```

```
     Left indices:  [0, 1, 2, 3, 4, 7, 9]
     Right indices:  [5, 6, 8]
      All tests passed.
```

```
# UNQ_C3
# GRADED FUNCTION: compute_information_gain

def compute_information_gain(X, y, node_indices, feature):

    """
    Compute the information of splitting the node on a given feature

    Args:
        X (ndarray):            Data matrix of shape(n_samples, n_features)
        y (array like):         list or ndarray with n_samples containing the target variable
        node_indices (ndarray): List containing the active indices. I.e, the samples being considered in this step.

    Returns:
        cost (float):        Cost computed

    """
    # Split dataset
    left_indices, right_indices = split_dataset(X, node_indices, feature)

    # Some useful variables
    X_node, y_node = X[node_indices], y[node_indices]
    X_left, y_left = X[left_indices], y[left_indices]
    X_right, y_right = X[right_indices], y[right_indices]

    # You need to return the following variables correctly
    information_gain = 0

    ### START CODE HERE ###
    node_entropy = compute_entropy(y_node)
    left_entropy = compute_entropy(y_left)
    right_entropy = compute_entropy(y_right)

    # Weights
    w_left = len(X_left) / len(X_node)
    w_right = len(X_right) / len(X_node)

    #Weighted entropy
    weighted_entropy = w_left * left_entropy + w_right * right_entropy

    #Information gain
    information_gain = node_entropy - weighted_entropy

    ### END CODE HERE ###

    return information_gain


info_gain0 = compute_information_gain(X_train, y_train, root_indices, feature=0)
print("Information Gain from splitting the root on brown cap: ", info_gain0)

info_gain1 = compute_information_gain(X_train, y_train, root_indices, feature=1)
print("Information Gain from splitting the root on tapering stalk shape: ", info_gain1)

info_gain2 = compute_information_gain(X_train, y_train, root_indices, feature=2)
print("Information Gain from splitting the root on solitary: ", info_gain2)

# UNIT TESTS
compute_information_gain_test(compute_information_gain)
```

```
     Information Gain from splitting the root on brown cap:  0.034851554559677034
     Information Gain from splitting the root on tapering stalk shape:  0.12451124978365313
     Information Gain from splitting the root on solitary:  0.2780719051126377
      All tests passed.
```

```
def get_best_split(X, y, node_indices):
    """
    Returns the optimal feature and threshold value
    to split the node data

    Args:
        X (ndarray):            Data matrix of shape(n_samples, n_features)
        y (array like):         list or ndarray with n_samples containing the target variable
tree = []

def build_tree_recursive(X, y, node_indices, branch_name, max_depth, current_depth):
    """
    Build a tree using the recursive algorithm that split the dataset into 2 subgroups at each node.
    This function just prints the tree.

    Args:
        X (ndarray):            Data matrix of shape(n_samples, n_features)
```