```
"""
Random Forest Lab

Name: Jay Cui
Section: 002
Date: 12/4/22
"""
import os
# import graphviz
# from uuid import uuid4
import numpy as np
from math import sqrt, floor
from sklearn.ensemble import RandomForestClassifier
import time

# Problem 1
class Question:
    """Questions to use in construction and display of Decision Trees.
    Attributes:
        column (int): which column of the data this question asks
        value (int/float): value the question asks about
        features (str): name of the feature asked about
    Methods:
        match: returns boolean of if a given sample answered T/F"""

    def __init__(self, column, value, feature_names):
        self.column = column
        self.value = value
        self.features = feature_names[self.column]

    def match(self, sample):
        """Returns T/F depending on how the sample answers the question
        Parameters:
            sample ((n,), ndarray): New sample to classify
        Returns:
            (bool): How the sample compares to the question"""
        return sample[self.column] >= self.value

    def __repr__(self):
        return "Is %s >= %s?" % (self.features, str(self.value))

def partition(data, question):
    """Splits the data into left (true) and right (false)
    Parameters:
        data ((m,n), ndarray): data to partition
        question (Question): question to split on
    Returns:
        left ((j,n), ndarray): Portion of the data matching the question
        right ((m-j, n), ndarray): Portion of the data NOT matching the question
    """
    return data[data[:, question.column] >= question.value, :], data[data[:, question.column]
< question.value, :]


# Helper function
def num_rows(array):
    """ Returns the number of rows in a given array """
    if array is None:
        return 0
    elif len(array.shape) == 1:
        return 1
    else:
        return array.shape[0]
```

```
# Helper function
def class_counts(data):
    """ Returns a dictionary with the number of samples under each class label
        formatted {label : number_of_samples} """
    if len(data.shape) == 1: # If there's only one row
        return {data[-1] : 1}
    counts = {}
    for label in data[:,-1]:
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts


#Problem 2
def gini(data):
    """Return the Gini impurity of given array of data.
    Parameters:
        data (ndarray): data to examine
    Returns:
        (float): Gini impurity of the data"""
    label_counts = {}
    for label in data[:, -1]:
        if label in label_counts:
            label_counts[label] += 1
        else:
            label_counts[label] = 1

    N = len(data)
    impurity = 1
    for count in label_counts.values():
        impurity -= (count / N)**2

    return impurity

def info_gain(left, right, G):
    """Return the info gain of a partition of data.
    Parameters:
        left (ndarray): left split of data
        right (ndarray): right split of data
        G (float): Gini impurity of unsplit data
    Returns:
        (float): info gain of the data"""
    N = len(left) + len(right)
    return G - len(left) / N * gini(left) - len(right) / N * gini(right)

# Problem 3, Problem 7
def find_best_split(data, feature_names, min_samples_leaf=5, random_subset=False):
    """Find the optimal split
    Parameters:
        data (ndarray): Data in question
        feature_names (list of strings): Labels for each column of data
        min_samples_leaf (int): minimum number of samples per leaf
        random_subset (bool): for Problem 7
    Returns:
        (float): Best info gain
        (Question): Best question"""
    best_gain = 0
    best_question = None
    features = feature_names[:-1]
    if random_subset:
        n = len(features)
        n_sqrt = floor(sqrt(n))
        indices = np.random.randint(low=0, high=len(features), size=n_sqrt)
    G = gini(data)
    for i in range(len(features)):
        if random_subset and i not in indices:
            continue
        unique_values = np.unique(data[:, i])
        for unique_value in unique_values:
            question = Question(column=i, value=unique_value, feature_names=features)
            left, right = partition(data, question)
            if len(left) < min_samples_leaf or len(right) < min_samples_leaf:
                continue
            I = info_gain(left, right, G)
            if I > best_gain:
                best_gain = I
                best_question = question

    return best_gain, best_question


# Problem 4
class Leaf:
    """Tree leaf node
    Attribute:
        prediction (dict): Dictionary of labels at the leaf"""
    def __init__(self,data):
        self.prediction = class_counts(data)

class Decision_Node:
    """Tree node with a question
    Attributes:
        question (Question): Question associated with node
        left (Decision_Node or Leaf): child branch
        right (Decision_Node or Leaf): child branch"""
    def __init__(self, question, left_branch, right_branch):
        self.question = question
        self.left = left_branch
        self.right = right_branch


# Prolem 5
def build_tree(data, feature_names, min_samples_leaf=5, max_depth=4, current_depth=0,
random_subset=False):
    """Build a classification tree using the classes Decision_Node and Leaf
    Parameters:
        data (ndarray)
        feature_names(list or array)
        min_samples_leaf (int): minimum allowed number of samples per leaf
        max_depth (int): maximum allowed depth
        current_depth (int): depth counter
        random_subset (bool): whether or not to train on a random subset of features
    Returns:
        Decision_Node (or Leaf)"""
    if len(data) < 2 * min_samples_leaf:
        return Leaf(data)

    optimal_gain, corresponding_question = find_best_split(data, feature_names,
random_subset=random_subset)
    if optimal_gain == 0 or current_depth >= max_depth:
        return Leaf(data)

    left, right = partition(data, corresponding_question)
    left_branch = build_tree(
        left,
        feature_names,
        min_samples_leaf=min_samples_leaf,
        max_depth=max_depth,
        current_depth=current_depth+1,
        random_subset=random_subset
    )
    right_branch = build_tree(
        right,
        feature_names,
        min_samples_leaf=min_samples_leaf,
        max_depth=max_depth,
        current_depth=current_depth+1,
        random_subset=random_subset
    )
    return Decision_Node(corresponding_question, left_branch, right_branch)

# Problem 6
def predict_tree(sample, my_tree):
    """Predict the label for a sample given a pre-made decision tree
    Parameters:
        sample (ndarray): a single sample
        my_tree (Decision_Node or Leaf): a decision tree
    Returns:
        Label to be assigned to new sample"""
    if isinstance(my_tree, Leaf):
        return max(my_tree.prediction, key=my_tree.prediction.get)

    if my_tree.question.match(sample):
        return predict_tree(sample, my_tree.left)
    else:
        return predict_tree(sample, my_tree.right)

def analyze_tree(dataset, my_tree):
    """Test how accurately a tree classifies a dataset
    Parameters:
        dataset (ndarray): Labeled data with the labels in the last column
        tree (Decision_Node or Leaf): a decision tree
    Returns:
        (float): Proportion of dataset classified correctly"""
    N = len(dataset)
    correct_count = 0
    for sample in dataset:
        correct_count += int(predict_tree(sample, my_tree) == sample[-1])

    return correct_count / N

# Problem 7
def predict_forest(sample, forest):
    """Predict the label for a new sample, given a random forest
    Parameters:
        sample (ndarray): a single sample
        forest (list): a list of decision trees
    Returns:
        Label to be assigned to new sample"""
    labels = [predict_tree(sample, tree) for tree in forest]
    return max(set(labels), key=labels.count)

def analyze_forest(dataset, forest):
    """Test how accurately a forest classifies a dataset
    Parameters:
        dataset (ndarray): Labeled data with the labels in the last column
        forest (list): list of decision trees
    Returns:
        (float): Proportion of dataset classified correctly"""
    N = len(dataset)
    correct_count = 0
    for sample in dataset:
        correct_count += int(predict_forest(sample, forest) == sample[-1])
    return correct_count / N

# Problem 8
def prob8():
    """ Using the file parkinsons.csv, return three tuples. For tuples 1 and 2,
        randomly select 130 samples; use 100 for training and 30 for testing.
        For tuple 3, use the entire dataset with an 80-20 train-test split.
        Tuple 1:
            a) Your accuracy in a 5-tree forest with min_samples_leaf=15
                and max_depth=4
            b) The time it took to run your 5-tree forest
        Tuple 2:
            a) Scikit-Learn's accuracy in a 5-tree forest with
                min_samples_leaf=15 and max_depth=4
            b) The time it took to run that 5-tree forest
        Tuple 3:
            a) Scikit-Learn's accuracy in a forest with default parameters
            b) The time it took to run that forest with default parameters
    """
    parkinsons = np.loadtxt('parkinsons.csv', delimiter=',', dtype=float, comments=None)
    parkinsons_features = np.loadtxt('parkinsons_features.csv', delimiter=',', dtype=str,
comments=None)
    parkinsons = parkinsons[:, 1:]
    parkinsons_features = parkinsons_features[1:]
    np.random.shuffle(parkinsons)
    training_subset, test_subset = parkinsons[:100, :], parkinsons[100:130, :]

    my_start_time = time.time()
    my_forest = [build_tree(
            data=training_subset[:, :-1],
            feature_names=parkinsons_features,
            min_samples_leaf=15,
            max_depth=4,
            random_subset=True
        ) for _ in range(5)]
    my_accuracy = analyze_forest(dataset=test_subset[:, :-1], forest=my_forest)
    my_end_time = time.time()

    sklearn_start_time = time.time()
    sklearn_forest = RandomForestClassifier(n_estimators=5, max_depth=4, min_samples_leaf=15)
    sklearn_forest.fit(training_subset[:, :-1], training_subset[:, -1])
    sklearn_accuracy = sklearn_forest.score(test_subset[:, :-1], test_subset[:, -1])
    sklearn_end_time = time.time()

    N = len(parkinsons)
    l = floor(0.8 * N)
    training_set, test_set = parkinsons[:l, :], parkinsons[l:, :]
    sklearn_whole_start_time = time.time()
    sklearn_whole_forest = RandomForestClassifier()
    sklearn_whole_forest.fit(training_set[:, :-1], training_set[:, -1])
    sklearn_whole_accuracy = sklearn_whole_forest.score(test_set[:, :-1], test_set[:, -1])
    sklearn_whole_end_time = time.time()

    return (f'{int(my_accuracy * 100)}%', f'{my_end_time - my_start_time} seconds'),
(f'{int(sklearn_accuracy * 100)}%', f'{sklearn_end_time - sklearn_start_time} seconds'),
(f'{int(sklearn_whole_accuracy * 100)}%', f'{sklearn_whole_end_time -
sklearn_whole_start_time} seconds')


## Code to draw a tree
def draw_node(graph, my_tree):
    """Helper function for drawTree"""
    node_id = uuid4().hex
    #If it's a leaf, draw an oval and label with the prediction
    if isinstance(my_tree, Leaf):
        graph.node(node_id, shape="oval", label="%s" % my_tree.prediction)
        return node_id
    else: #If it's not a leaf, make a question box
        graph.node(node_id, shape="box", label="%s" % my_tree.question)
        left_id = draw_node(graph, my_tree.left)
        graph.edge(node_id, left_id, label="T")
        right_id = draw_node(graph, my_tree.right)
        graph.edge(node_id, right_id, label="F")
        return node_id

def draw_tree(my_tree):
    """Draws a tree"""
    #Remove the files if they already exist
    for file in ['Digraph.gv','Digraph.gv.pdf']:
        if os.path.exists(file):
            os.remove(file)
    graph = graphviz.Digraph(comment="Decision Tree")
    draw_node(graph, my_tree)
    graph.render(view=True) #This saves Digraph.gv and Digraph.gv.pdf


if __name__=='__main__':
    '''Unit tests.'''
    # Problem 1
    question = Question(column=0, value=0, feature_names=['1st column'])

    np.set_printoptions(suppress=True)
    animals = np.loadtxt('animals.csv', delimiter=',')
    left, right = partition(animals, question)

    # Problem 2
    animal_features = np.loadtxt('animal_features.csv', delimiter=',', dtype=str,
comments=None)
    animal_names = np.loadtxt('animal_names.csv', delimiter=',', dtype=str)
    gini_animals = gini(animals)
    assert gini_animals == 0.4758
    assert info_gain(animals[:50], animals[50:], gini_animals) == 0.1458

    # Problem 3
    print(find_best_split(animals, animal_features)) # FIXME: value should be 2.0, not 0.0

    # Problem 4, 5, 6
    np.random.shuffle(animals)
    training_set, test_set = animals[:80, :], animals[80:, :]
    my_tree = build_tree(data=training_set, feature_names=animal_features)
    print(f'Decision tree accuracy: {analyze_tree(dataset=test_set, my_tree=my_tree)}')

    # Problem 7
    my_forest = [build_tree(data=training_set, feature_names=animal_features,
random_subset=True) for _ in range(10)]
    print(f'Random forest accuracy: {analyze_forest(dataset=test_set, forest=my_forest)}')

    # Problem 8
    print(prob8())
```