```python
import pandas as pd
import numpy as np
from pprint import pprint
import sys

# Reads the data from CSV files, each attribute column can be obtained via its name, e.g., y = data['y']
def getDataframe(filePath):
    data = pd.read_csv(filePath)
    return data

# predicted_y and y are the predicted and actual y values respectively as numpy arrays
# function prints the accuracy
def compute_accuracy(predicted_y, y):
    acc = 100.0
    acc = np.sum(predicted_y == y)/predicted_y.shape[0]
    return acc

#Compute entropy according to y distribution
def compute_entropy(y):
    entropy = 0.0
    elements,counts = np.unique(y, return_counts = True)
    n = y.shape[0]

    for i in range(len(elements)):
        prob = counts[i]/n
        if prob!= 0:
            entropy -= prob * np.log2(prob)
    return entropy

#att_name: attribute name; y_name: the target attribute name for classification
def compute_info_gain(data, att_name, y_name):
    info_gain = 0.0

    #Calculate the values and the corresponding counts for the select attribute
    vals, counts = np.unique(data[att_name], return_counts=True)
    total_counts = np.sum(counts)
    #Calculate the conditional entropy
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    total_info = compute_entropy(data[y_name])
    info_A = 0.0
    for i in range(len(vals)):
        info_A += (counts[i]/total_counts) * compute_entropy(data.loc[data[att_name] == vals[i]][y_name])
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    info_gain = total_info - info_A
    return info_gain


def comput_gain_ratio(data, att_name, y_name):
    gain_ratio = 0.0
    #Calculate the values and the corresponding counts for the select
```

```python
    vals, counts = np.unique(data[att_name], return_counts=True)
    total_counts = np.sum(counts)

    #Calculate the information for the selected attribute
    att_info = 0.0
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    for i in range(len(vals)):
        p = counts[i] / total_counts
        att_info -= p * np.log2(p)
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    gain_ratio = 0.0 if np.abs(att_info) < 1e-9 else min(1, compute_info_gain(data, att_name, y_name) / att_info)
    return gain_ratio

# Class of the decision tree model based on the ID3 algorithm
class DecisionTree(object):
    def __init__(self):
        self.train_data = pd.DataFrame()
        self.test_data = pd.DataFrame()

    def load_data(self, train_file, test_file):
        self.train_data = getDataframe(train_file)
        self.test_data = getDataframe(test_file)

    def train(self, y_name, measure, parent_node_class = None):
        self.y_name = y_name
        self.measure = measure
        self.tree = self.make_tree(self.train_data, parent_node_class)

    def make_tree(self, train_data, parent_node_class = None):
        data = train_data
        features = data.drop(self.y_name, axis = 1).columns.values
        measure =  self.measure
        #Stopping condition 1: If all target_values have the same value, return this value
        if len(np.unique(data[self.y_name])) <= 1:
            leaf_value = -1
            #========================#
            # STRART YOUR CODE HERE  #
            #========================#
            leaf_node = np.unique(data[self.y_name])[0]
            #========================#
            #   END YOUR CODE HERE   #
            #========================#
            return leaf_node

        #Stopping condition 2: If the dataset is empty, return the parent_node_class
        elif len(data)== 0:
            return parent_node_class

        #Stopping condition 3: If the feature space is empty, return the majority class
        elif len(features) == 0:
            return np.unique(data[self.y_name])
```

```python

        # Not a leaf node, create an internal node
        else:
            #Set the default value for this node --> The mode target feature value of the current node
            parent_node_class = np.unique(data[self.y_name])[np.argmax(np.unique(data[self.y_name],return_counts=True)[1])]

            #Select the feature which best splits the dataset
            if measure == 'info_gain':
                item_values = [compute_info_gain(data, feature, self.y_name) for feature in features] #Return the information gain values for the features in the dataset
            elif measure == 'gain_ratio':
                item_values = [comput_gain_ratio(data, feature, self.y_name) for feature in features] #Return the gain_ratio for the features in the dataset
            else:
                raise ValueError("kernel not recognized")

            best_feature_index = np.argmax(item_values)
            best_feature = features[best_feature_index]
            print('best_feature is: ', best_feature)

            #Create the tree structure. The root gets the name of the feature (best_feature)
            tree = {best_feature:{}}


        #Grow a branch under the root node for each possible value of the root node feature

        for value in np.unique(data[best_feature]):
            #Split the dataset along the value of the feature with the largest information gain and therwith create sub_datasets
            sub_data = data.where(data[best_feature] == value).dropna()

            #Remove the selected feature from the feature space
            sub_data = sub_data.drop(best_feature, axis = 1)

            #Call the ID3 algorithm for each of those sub_datasets with the new parameters --> Here the recursion comes in!
            subtree = self.make_tree(sub_data, parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the tree under the root node
            tree[best_feature][value] = subtree

        return tree


    def test(self, y_name):
        accuracy = self.classify(self.test_data, y_name)
        return accuracy

    def classify(self, test_data, y_name):
        #Create new query instances by simply removing the target feature column from the test dataset and
        #convert it to a dictionary
```

```
        test_y = test_data[y_name]

        n = test_data.shape[0]
        predicted_y = np.zeros(n)

        #Calculate the prediction accuracy
        for i in range(n):
            predicted_y[i] = DecisionTree.predict(self.tree, test_x.iloc[i])

        output = np.zeros((n,2))
        output[:,0] = test_y
        output[:,1] = predicted_y
        accuracy = compute_accuracy(predicted_y, test_y.values)
        return accuracy

    def predict(tree, query):
        # find the root attribute
        default = -1
        for root_name in list(tree.keys()):
            try:
                subtree = tree[root_name][query[root_name]]
            except:
                return default ## root_name does not appear in query attribute list (it is an error!)

            ##if subtree is still a dictionary, recursively test next attribute
            if isinstance(subtree,dict):
                return DecisionTree.predict(subtree, query)
            else:
                leaf = subtree
                return leaf


```