```
import numpy as np
import pandas as pd


# hàm chia node thành 2 node con dựa trên ngưỡng
def split_node(column, threshold_split):  # column là series
    left_node = column[column <= threshold_split].index  # index của các phần tử nhỏ hơn hoặc bằng ngưỡng
    right_node = column[column > threshold_split].index  # index của các phần tử lớn hơn ngưỡng
    return left_node, right_node # chứa giá trị index


# hàm tính entropy
def entropy(y_target):  # y_target là dạng series
    values, counts = np.unique(y_target, return_counts = True) # biến values chưa được dùng
    result = -np.sum([(count / len(y_target)) * np.log2(count / len(y_target)) for count in counts])
    return result  # kết quả là một số


# hàm tính information gain
def info_gain(column, target, threshold_split):  # column, target là series, threshold_split là một số
    entropy_start = entropy(target)  # entropy ban đầu

    left_node, right_node = split_node(column, threshold_split)  # chia dữ liệu thành 2 node con

    n_target = len(target)  # số lượng mẫu trong target
    n_left = len(left_node)  # số lượng mẫu ở node trái
    n_right = len(right_node)  # số lượng mẫu ở node phải

    # tính entropy cho các node con
    entropy_left = entropy(target[left_node])  # target[left_node] là series
    entropy_right = entropy(target[right_node]) # target[right_node] là series

    # Tính tổng entropy của các node con
    weight_entropy = (n_left / n_target) * entropy_left + (n_right / n_target) * entropy_right

    # Tính Information Gain
    ig = entropy_start - weight_entropy
    return ig


# hàm tìm feature và threshold tốt nhất để chia
def best_split(dataX, target, feature_id):  # dataX dạng DataFrame, target dạng series
    best_ig = -1  # khởi tạo ig tốt nhất là trừ vô cùng hoặc một con số nhỏ hơn 0 là được
    best_feature = None # best_feature, best_threshold không cần so sánh nên ta gán giá trị là None
    best_threshold = None
    for _id in feature_id:
        column = dataX.iloc[:, _id] # dạng series
        thresholds = set(column)
        for threshold in thresholds:  # duyệt qua từng giá trị threshold
            ig = info_gain(column, target, threshold) # threshold là một con số
            if ig > best_ig: # xét điều kiện nếu ig tính ra lớn hơn best_ig thì
                best_ig = ig # gán best_ig bằng ig
                best_feature = dataX.columns[_id] # gán best feature
                best_threshold = threshold # gán lại ngưỡng
    return best_feature, best_threshold  # trả về feature và threshold tốt nhất


# hàm lấy giá trị xuất hiện nhiều nhất trong node lá
def most_value(y_target):  # y_target là series
    value = y_target.value_counts().idxmax()  # giá trị xuất hiện nhiều nhất
    return value # trả lại giá trị của node lá


# lớp Node đại diện cho từng node trong cây
class Node:
    def __init__(self, feature = None, threshold = None, left = None, right = None, *, value = None): # những tham số sau '*' cần khai báo r
        self.feature = feature    # feature để chia node
        self.threshold = feature    # ngưỡng để chia node
        self.left =  feature   # node con bên trái
        self.right = right     # node con bên phải
        self.value = value   # giá trị của node nếu là node lá

    def is_leaf_node(self): # hàm kiểm tra có phải là node lá hay không
        return   # nếu có value, tức là node lá


# lớp Decision Tree Classification
class DecisionTreeClass:
```

```
    def __init__(self, min_samples_split = 2, max_depth = 10, n_features = None):
        self.min_samples_split = min_samples_split    # số lượng mẫu tối thiểu để chia một nút
        self.max_depth = max_depth   # độ sâu tối đa của cây
        self.root = None  # node gốc của cây
        self.n_features = n_features    # số cột cần lấy để tạo cây
    def grow_tree(self, X, y, depth = 0):  # X là frame, y là series
        n_samples, n_feats =  X.shape   # số lượng mẫu và số lượng đặc trưng
        n_classes = len(np.unique(y))  # số lượng lớp phân loại khác nhau

        # Điều kiện dừng: nếu đạt độ sâu tối đa hoặc không thể chia thêm (số lớp trong node bằng 1 hoặc số mẫu trong node nhỏ hơn số lượng m
        if depth >= self.max_depth or n_classes == 1 or n_samples < self.min_samples_split:
            leaf_value = most_value(y)
            return Node(value = leaf_value) # lúc này node có value khác None nên đây là node lá

        # lấy số cột ngẫu nhiên khi tham số n_features khác None
        feature_id = np.random.choice(n_feats, self.n_features, replace = False)

        # tìm feature và threshold tốt nhất để chia
        best_feature, best_threshold = best_split(X, y, feature_id)

        # tách node thành node trái và phải
        left_node, right_node = split_node(X[best_feature], best_threshold)

        # dùng đệ quy để xây dựng cây con
        # phải sử dụng loc, không sử dụng iloc vì lúc này index là label
        # các bạn phải hiểu sự khác biệt giữa iloc và loc
        left = self.grow_tree(X.loc[left_node], y.loc[left_node], depth + 1)
        right = self.grow_tree(X.loc[right_node], y.loc[right_node], depth + 1)

        # trả về node hiện tại với thông tin chia và 2 node con
        return Node(best_feature, best_threshold, left, right)

    def fit(self, X, y):  # X là frame, y là series
        # nếu n_features là None, tức nghĩa là người dùng không truyền giá trị vào thì lấy tất cả các feature đang có
        # ngược lại, nếu người dùng truyền vào giá trị thì lấy số cột(giá trị) người dùng truyền vào
        self.n_features = X.shape[1] if self.n_features is None else min(X.shape[1], self.n_features)
        self.root = self.grow_tree(X, y)  # gọi hàm xây dựng cây

    def traverse_tree(self, x, node):  # hàm duyệt cây để dự đoán, x là series
        # ý tưởng của nó là duyệt đến khi tìm được nút lá bằng cách gọi đệ quy
        if node.is_leaf_node():
            return node.value

        if x[node.feature] <= node.threshold:
            return self.traverse_tree(x, node.left)
        return self.traverse_tree(x, node.right)

    def predict(self, X):  # hàm dự đoán cho tất cả mẫu trong X, lấy từng dòng trong X để duyệt
        return np.array([self.traverse_tree(x, self.root) for index, x in X.iterrows()]) # index không sử dụng, x là dạng Series


# hàm vẽ cây
def print_tree(node, indent = ""):
    # nếu node là node lá, in ra giá trị của nó
    if node.is_leaf_node():
        print(f"{indent}Leaf: {node.value}")
        return

    # in ra node hiện tại với feature và threshold
    print(f"{indent}Node: If {node.feature} <= {node.threshold:.2f}")

    # in ra cây con bên trái (đúng)
    print(f"{indent}  True:")
    print_tree(node.left, indent + "    ")

    # in ra cây con bên phải (sai)
    print(f"{indent}  False:")
    print_tree(node.right, indent + "    ")


# hàm tính độ chính xác
def accuracy(y_actual, y_pred): # hai này ở dạng mảng
    acc = np.sum(y_actual == y_pred) / len(y_actual)
    return acc*100
```

Start coding or generate with AI.