```python
import numpy as np

import pandas as pd

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels (Setosa, Versicolor, Virginica)
print(x)
print(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

def entropy(y):
    unique_classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    return -np.sum(probabilities * np.log2(probabilities))

def information_gain(X, y, feature_index, threshold):
    parent_entropy = entropy(y)

    left_indices = X[:, feature_index] <= threshold
    right_indices = X[:, feature_index] > threshold

    n, n_left, n_right = len(y), np.sum(left_indices), np.sum(right_indices)

    if n_left == 0 or n_right == 0:  # Avoid splitting into empty groups
        return 0

    left_entropy = entropy(y[left_indices])
    right_entropy = entropy(y[right_indices])

    weighted_entropy = (n_left / n) * left_entropy + (n_right / n) * right_entropy

    return parent_entropy - weighted_entropy


def best_split(X, y):
    best_gain = 0
    best_feature = None
    best_threshold = None

    for feature_index in range(X.shape[1]):  # Iterate through features
```

```
        thresholds = np.unique(X[:, feature_index])  # Unique values in feature

        for threshold in thresholds:
            gain = information_gain(X, y, feature_index, threshold)

            if gain > best_gain:
                best_gain = gain
                best_feature = feature_index
                best_threshold = threshold

    return best_feature, best_threshold


class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

def build_tree(X, y, depth=0, max_depth=5):
    if len(np.unique(y)) == 1:  # If all samples belong to one class
        return Node(value=y[0])

    if depth >= max_depth:
        unique_classes, counts = np.unique(y, return_counts=True)
        return Node(value=unique_classes[np.argmax(counts)])

    feature, threshold = best_split(X, y)

    if feature is None:
        unique_classes, counts = np.unique(y, return_counts=True)
        return Node(value=unique_classes[np.argmax(counts)])

    left_indices = X[:, feature] <= threshold
    right_indices = X[:, feature] > threshold

    left_subtree = build_tree(X[left_indices], y[left_indices], depth + 1, max_depth)
    right_subtree = build_tree(X[right_indices], y[right_indices], depth + 1, max_depth)

    return Node(feature, threshold, left_subtree, right_subtree)



def predict_one(node, x):
    if node.value is not None:
        return node.value  # Return class label for leaf node
```

```
    if x[node.feature] <= node.threshold:
        return predict_one(node.left, x)
    else:
        return predict_one(node.right, x)

def predict(tree, X):
    return np.array([predict_one(tree, x) for x in X])


# Train the Decision Tree
tree = build_tree(X_train, y_train)

# Make Predictions
y_pred = predict(tree, X_test)

# Evaluate Model Performance
accuracy = accuracy_score(y_test, y_pred)
print("Decision Tree Accuracy:", accuracy)


def print_tree(node, depth=0):
    if node.value is not None:  # Leaf node
        print("  " * depth + f"Leaf: Class {node.value}")
        return

    # Print feature and threshold
    print("  " * depth + f"Feature {node.feature} <= {node.threshold}?")

    # Print left and right subtree
    print("  " * depth + "Left:")
    print_tree(node.left, depth + 1)

    print("  " * depth + "Right:")
    print_tree(node.right, depth + 1)

# Print the trained decision tree
print_tree(tree)
```