```
from collections import Counter
import numpy as np
class Node:
    def __init__(self, feature=None, value=None, results=None, true_branch=None, false_branch=None):
        self.feature = feature  # Feature to split on
        self.value = value      # Value of the feature to split on
        self.results = results  # Stores class labels if node is a leaf node
        self.true_branch = true_branch  # Branch for values that are True for the feature
        self.false_branch = false_branch  # Branch for values that are False for the feature
def entropy(data):
    counts = np.bincount(data)
    probabilities = counts / len(data)
    entropy = -np.sum([p * np.log2(p) for p in probabilities if p > 0])
    return entropy
def split_data(X, y, feature, value):
    true_indices = np.where(X[:, feature] <= value)[0]
    false_indices = np.where(X[:, feature] > value)[0]
    true_X, true_y = X[true_indices], y[true_indices]
    false_X, false_y = X[false_indices], y[false_indices]
    return true_X, true_y, false_X, false_y
def build_tree(X, y):
    if len(set(y)) == 1:
        return Node(results=y[0])

    best_gain = 0
    best_criteria = None
    best_sets = None
    n_features = X.shape[1]

    current_entropy = entropy(y)

    for feature in range(n_features):
        feature_values = set(X[:, feature])
        for value in feature_values:
            true_X, true_y, false_X, false_y = split_data(X, y, feature, value)
            true_entropy = entropy(true_y)
            false_entropy = entropy(false_y)
            p = len(true_y) / len(y)
            gain = current_entropy - p * true_entropy - (1 - p) * false_entropy

            if gain > best_gain:
                best_gain = gain
                best_criteria = (feature, value)
```

```
                best_sets = (true_X, true_y, false_X, false_y)

    if best_gain > 0:
        true_branch = build_tree(best_sets[0], best_sets[1])
        false_branch = build_tree(best_sets[2], best_sets[3])
        return Node(feature=best_criteria[0], value=best_criteria[1], true_branch=true_branch, false_branch=false_branch)

    return Node(results=y[0])
def predict(tree, sample):
    if tree.results is not None:
        return tree.results
    else:
        branch = tree.false_branch
        if sample[tree.feature] <= tree.value:
            branch = tree.true_branch
        return predict(branch, sample)
X = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([1, 1, 0, 0])

# Building the tree
decision_tree = build_tree(X, y)
X = np.array([[1, 1], [1, 0], [0, 1], [0, 0]])
y = np.array([1, 1, 0, 0])

# Building the tree
decision_tree = build_tree(X, y)
sample = np.array([1, 0])
prediction = predict(decision_tree, sample)
print(f"Prediction for sample {sample}: {prediction}")
```

```
Prediction for sample [1 0]: 1
```