```
#Import
import numpy as np
import pandas as pd

#Getting the data
col_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
data = pd.read_csv("iris.csv", skiprows=1, header=None, names=col_names)
data.head(10)

#Node Class
class Node():
  def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=
    '''constructor'''

    #for decision node
    self.feature_index = feature_index
    self.threshold = threshold
    self.left = left
    self.right = right
    self.info_gain = info_gain

    #for leaf node
    self.value = value

#Tree Class
class DecisionTreeClassifier():
  def __init__(self, min_samples_split=2, max_depth=2):
    '''constructor'''

    #initialize root of the tree
    self.root = None

    #stopping conditions
    self.min_samples_split = min_samples_split
    self.max_depth = max_depth

  def built_tree(self, dataset, curr_depth=0):
    '''recursive func. to built tree'''

    X, Y = dataset[:,:-1], dataset[:,-1]
    num_samples, num_features = np.shape(X)

    #split until stopping conditions are met
    if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
      #find best split
      best_split = self.get_best_split(dataset, num_samples, num_features)
      #check if information gain is positive
      if best_split["info_gain"]>0:
        #recur left
        left_subtree = self.built_tree(best_split["dataset_left"], curr_depth+1)
        #recur right
        right_subtree = self.built_tree(best_split["dataset_right"], curr_depth+1)
        #return decision node
        return Node(best_split["feature_index"], best_split["threshold"], left_subtree, ri
```

```
        return Node(best_split["feature_index"], best_split["threshold"], left_subtree, ri

      #compute leaf node
      leaf_value = self.calculate_leaf_value(Y)
      #return leaf node
      return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
      '''function to find the best split'''

      #dictonary to store the best split
      best_split = {}
      max_info_gain = -float("inf")

      #loop over all features
      for feature_index in range(num_features):
        feature_values = dataset[:, feature_index]
        possible_thresholds = np.unique(feature_values)
        #loop over all the features values present in the data
        for threshold in possible_thresholds:
          #get current split
          dataset_left, dataset_right = self.split(dataset, feature_index, threshold)
          #check if childs are not null
          if len(dataset_left)>0 and len(dataset_right)>0:
            y, left_y, right_y = dataset[:, -1], dataset_left[:, -1], dataset_right[:, -1]
            #compute information gain
            curr_info_gain = self.information_gain(y, left_y, right_y, "gini")
            #update the best split if needed
            if curr_info_gain>max_info_gain:
              best_split["feature_index"] = feature_index
              best_split["threshold"] = threshold
              best_split["dataset_left"] = dataset_left
              best_split["dataset_right"] = dataset_right
              best_split["info_gain"] = curr_info_gain
              max_info_gain = curr_info_gain

      #return best split
      return best_split

    def split(self, dataset, feature_index, threshold):
      '''function to split the data'''

      dataset_left = np.array([row for row in dataset if row[feature_index]<=threshold])
      dataset_right = np.array([row for row in dataset if row[feature_index]>threshold])
      return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
      '''function to compute information gain'''

      weight_l = len(l_child) / len(parent)
      weight_r = len(r_child) / len(parent)
      if mode=="gini":
        gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child) + weight_r*self.
      else:
        gain = self.entropy(parent) - (weight_l*self._entropy(l_child) + weight_r*self.entro
      return gain
```

```
    return gain

  def entropy(self, y):
    '''function to compute entropy'''

    class_labels = np.unique(y)
    entropy = 0
    for cls in class_labels:
      p_cls = len(y[y == cls]) / len(y)
      entropy += -p_cls * np.log2(p_cls)
    return entropy

  def gini_index(self, y):
    '''function to compute gini index'''

    class_labels = np.unique(y)
    gini = 0
    for cls in class_labels:
      p_cls = len(y[y == cls]) / len(y)
      gini += p_cls**2
    return 1 - gini

  def calculate_leaf_value(self, Y):
    '''function to calculate leaf node'''

    Y = list(Y)
    return max(Y, key=Y.count)

  def print_tree(self, tree=None, indent=" "):
    '''function to print tree'''

    if not tree:
      tree = self.root

    if tree.value is not None:
      print(tree.value)

    elif tree.feature_index == 0:
      print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
      print("%sleft:" % (indent), end="")
      self.print_tree(tree.left, indent + indent)
      print("%sright:" % (indent), end="")
      self.print_tree(tree.right, indent + indent)

    elif tree.feature_index == 1:
      print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
      print("%sleft:" % (indent), end="")
      self.print_tree(tree.left, indent + indent)
      print("%sright:" % (indent), end="")
      self.print_tree(tree.right, indent + indent)

    elif tree.feature_index == 2:
      print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
      print("%sleft:" % (indent), end="")
      self.print_tree(tree.left, indent + indent)
      print("%sright:" % (indent), end="")
```

```
      self.print_tree(tree.right, indent + indent)

    elif tree.feature_index == 3:
      print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.info_gain)
      print("%sleft:" % (indent), end="")
      self.print_tree(tree.left, indent + indent)
      print("%sright:" % (indent), end="")
      self.print_tree(tree.right, indent + indent)

  def print_rule(self, tree=None, indent="", rule=""):
    if not tree:
      tree = self.root

    if tree.value is not None:
      print(f"{rule} then class is {tree.value}")

    else:
      rule += f"if {col_names[tree.feature_index]} <= {tree.threshold} and "
      self.print_rule(tree.left,indent+indent,rule)

      for idx in range(len(rule)-1, -1, -1):
        if(rule[idx]=='<'):
          rule=rule[0:idx]+'>'+rule[idx+2:]
          break

      self.print_rule(tree.right,indent+indent,rule)

  def fit(self, X, Y):
    '''function to train the tree'''

    dataset = np.concatenate((X, Y), axis=1)
    self.root = self.built_tree(dataset)

  def predict(self, X):
    '''function to predict new dataset'''

    preditions = [self.make_prediction(x, self.root) for x in X]
    return preditions

  def make_prediction(self, x, tree):
    '''function to predict a single data point'''

    if tree.value!=None: return tree.value
    feature_val = x[tree.feature_index]
    if feature_val<=tree.threshold:
      return self.make_prediction(x, tree.left)
    else:
      return self.make_prediction(x, tree.right)

#Train-Test split
X = data.iloc[:, :-1].values
Y = data.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2, random_state=41)
```

```
#Fit the model
classifier = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train,Y_train)
classifier.print_tree()
classifier.print_rule()
```

```
X_2 <= 1.9 ? 0.337413853727144944
 left:Iris-setosa
 right:X_3 <= 1.5 ? 0.427106638180289
  left:X_2 <= 4.9 ? 0.05124653739612173
    left:Iris-versicolor
    right:Iris-virginica
  right:X_2 <= 5.0 ? 0.019631171921475288
    left:X_1 <= 2.8 ? 0.20833333333333334
        left:Iris-virginica
        right:Iris-versicolor
    right:Iris-virginica
if petal_length <= 1.9 and  then class is Iris-setosa
if petal_length > 1.9 and if petal_width <= 1.5 and if petal_length <= 4.9 and  then
if petal_length > 1.9 and if petal_width <= 1.5 and if petal_length > 4.9 and  then c
if petal_length > 1.9 and if petal_width > 1.5 and if petal_length <= 5.0 and if sepa
if petal_length > 1.9 and if petal_width > 1.5 and if petal_length <= 5.0 and if sepa
if petal_length > 1.9 and if petal_width > 1.5 and if petal_length > 5.0 and  then cl
```

0s completed at 21:09