Import tools

```
import numpy as np
import pandas as pd
```

Get the data

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

	sepal_length	sepal_width	petal_length	petal_width	type
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5.0	3.4	1.5	0.2	0
8	4.4	2.9	1.4	0.2	0
9	4.9	3.1	1.5	0.1	0

Next steps: View recommended plots

Node class

```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None, right=None, info_gain=None, value=None):
        ''' 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 the root of the tree
        self.root = None

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

    def build_tree(self, dataset, curr_depth=0):
        ''' recursive function to build the tree '''
```

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

        # stopping conditions
        if num_samples >= self.min_samples_split and curr_depth <= self.max_depth:
            # Find the best split
            best_split = self.get_best_split(dataset, num_samples, num_features)
            if best_split["info_gain"] > 0:
                # Recur left
                left_subtree = self.build_tree(best_split["dataset_left"], curr_depth + 1)
                # Recur right
                right_subtree = self.build_tree(best_split["dataset_right"], curr_depth + 1)
                # Return decision node
                return Node(
                    feature_index=best_split["feature_index"],
                    threshold=best_split["threshold"],
                    left=left_subtree,
                    right=right_subtree,
                    info_gain=best_split["info_gain"]
                )

        # 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 '''
        best_split = {}
        max_info_gain = -float("inf")

        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            for threshold in feature_values:
                # Split the dataset
                dataset_left, dataset_right = self.split(dataset, feature_index, threshold)

                if len(dataset_left) > 0 and len(dataset_right) > 0:
                    # Calculate information gain
                    info_gain = self.information_gain(dataset, dataset_left, dataset_right)

                    if info_gain > max_info_gain:
                        max_info_gain = info_gain
                        best_split = {
                            "feature_index": feature_index,
                            "threshold": threshold,
                            "dataset_left": dataset_left,
                            "dataset_right": dataset_right,
                            "info_gain": info_gain
                        }

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

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

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

        if not tree:
            tree = self.root

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

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

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

    def entropy(self, y):
        ''' function to compute entropy '''
        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = np.count_nonzero(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 = np.count_nonzero(y == cls) / len(y)
            gini += p_cls * (1 - p_cls)
        return gini

    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[:, -1]) - (weight_l * self.gini_index(l_child[:, -1]) + weight_r * self.gini_index(r_child[:, -1]))
        else:
            gain = self.entropy(parent[:, -1]) - (weight_l * self.entropy(l_child[:, -1]) + weight_r * self.entropy(r_child[:, -1]))
        return gain
```

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)
```

Test the model

```
Y_pred = classifier.predict(X_test)
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(Y_test, Y_pred)
print("Accuracy:", accuracy*100)
```

```
Accuracy: 93.33333333333333
```