Modify the Decision Tree scratch code in our lecture such that:

- Modify the scratch code so it can accept an hyperparameter `max_depth`, in which it will continue create the tree until max_depth is reached.
- Put everything into a class `DecisionTree`. It should have at least two methods, `fit()`, and `predict()`
- Load the iris data and try with your class

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
print("Imported")
```

```
Imported
```

In [2]:
```
#To help with our implementation, we create a class Node
class Node:
    def __init__(self, gini, num_samples, num_samples_per_class, predicted_class):
        self.gini = gini
        self.num_samples = num_samples
        self.num_samples_per_class = num_samples_per_class
        self.predicted_class = predicted_class
        self.feature_index = 0
        self.threshold = 0
        self.left = None
        self.right = None
```

In [48]:
```
class DecisionTree():

    def __init__(self, max_depth = None):
        self.max_depth = max_depth

    def node(self, gini, num_samples, num_samples_per_class, predicted_class):
        self.gini = gini
        self.num_samples = num_samples
        self.num_samples_per_class = num_samples_per_class
        self.predicted_class = predicted_class
        self.feature_index = 0
        self.threshold = 0
        self.left = None
        self.right = None

    def find_split(self,X, y, n_classes):
        """ Find split where children has lowest impurity possible
        in condition where the purity should also be less than the parent,
        if not, stop.
        """
        n_samples, n_features = X.shape
        if n_samples <= 1:
            return None, None

        #so it will not have any wasning about "referenced before assignments"
        feature_ix, threshold = None, None

        # Count of each class in the current node.
        sample_per_class_parent = [np.sum(y == c) for c in range(n_classes)] #[2, 2]

        # Gini of parent node.
        best_gini = 1.0 - sum((n / n_samples) ** 2 for n in sample_per_class_parent)

        # Loop through all features.
        for feature in range(n_features):

            # Sort data along selected feature.
            sample_sorted = sorted(X[:, feature]) #[2, 3, 10, 19]
            sort_idx = np.argsort(X[:, feature])
            y_sorted = y[sort_idx] #[0, 0, 1, 1]

            sample_per_class_left = [0] * n_classes   #[0, 0]

            sample_per_class_right = sample_per_class_parent.copy() #[2, 2]

            #loop through each threshold, 2.5, 6.5, 14.5
            #1st iter: [-] [++]
            #2nd iter: [--] [++]
            #3rd iter: [---] [+]
            for i in range(1, n_samples): #1 to 3 (excluding 4)
                # the class of that sample
                c = y_sorted[i - 1]  #[0]

                #put the sample to the left
                sample_per_class_left[c] += 1  #[1, 0]

                #take the sample out from the right  [1, 2]
                sample_per_class_right[c] -= 1

                gini_left = 1.0 - sum(
                    (sample_per_class_left[x] / i) ** 2 for x in range(n_classes)
                )

                #we divided by n_samples - i since we know that the left amount of samples
                #since left side has already i samples
                gini_right = 1.0 - sum(
                    (sample_per_class_right[x] / (n_samples - i)) ** 2 for x in range(n_classes)
                )

                #weighted gini
                weighted_gini = ((i / n_samples) * gini_left) + ( (n_samples - i) /n_samples) * gini_ri
ght

                # in case the value are the same, we do not split
                # (both have to end up on the same side of a split).
                if sample_sorted[i] == sample_sorted[i - 1]:
                    continue

                if weighted_gini < best_gini:
                    best_gini = weighted_gini
                    feature_ix = feature
                    threshold = (sample_sorted[i] + sample_sorted[i - 1]) / 2  # midpoint

        #return the feature number and threshold
        #used to find best split
        return feature_ix, threshold

    def fit(self, Xtrain, ytrain, n_classes, depth=0):
        n_samples, n_features = Xtrain.shape
        num_samples_per_class = [np.sum(ytrain == i) for i in range(n_classes)]
        #predicted class using the majority of sample class
        predicted_class = np.argmax(num_samples_per_class)

        #define the parent node
        node = Node(
            gini = 1 - sum((np.sum(ytrain == c) / n_samples) ** 2 for c in range(n_classes)),
            predicted_class=predicted_class,
            num_samples = ytrain.size,
            num_samples_per_class = num_samples_per_class,
            )

        #perform recursion
        feature, threshold = self.find_split(Xtrain, ytrain, n_classes)
        if feature is not None:
            #take all the indices that is less than threshold
            indices_left = Xtrain[:, feature] < threshold
            X_left, y_left = Xtrain[indices_left], ytrain[indices_left]

            #tilde for negation
            X_right, y_right = Xtrain[~indices_left], ytrain[~indices_left]

            #take note for later decision
            node.feature_index = feature
            node.threshold = threshold

            if self.max_depth is not None and depth > self.max_depth:
                return node

            node.left = self.fit(X_left, y_left, n_classes, depth + 1)
            node.right = self.fit(X_right, y_right, n_classes, depth + 1)
            self.tree = node
        return node

    #
    def predict(self, sample):
        node = self.tree
        while node.left:
            if sample[node.feature_index] < node.threshold:
                node = node.left
            else:
                node = node.right
        return node.predicted_class
```

In [49]:
```
# fit starting with tree depth = 0
Xtrain = np.array([[2, 5],[3, 5],[10, 5],[19, 5]])
ytrain = np.array([0, 0, 1, 1])
Xtest = np.array([[4, 6],[6, 8],[9, 2],[12, 8]])
ytest = np.array([0, 0, 1, 1])

model = DecisionTree(max_depth = 0)


tree = model.fit(Xtrain, ytrain, len(set(ytrain)))
pred = [model.predict(x) for x in Xtest]

print("Tree feature ind: ", tree.feature_index)
print("Tree threshold: ", tree.threshold)
print("Pred: ", np.array(pred))
print("ytest: ", ytest)
```

```
Tree feature ind:  0
Tree threshold:  6.5
Pred:  [0 0 1 1]
ytest:  [0 0 1 1]
```

In [75]:
```
# Load iris
from sklearn.datasets import load_iris
data = load_iris()

X = data.data
y = data.target
```

In [76]:
```
import pandas as pd

# Check stuff
df = pd.DataFrame(X)
df
```

Out[76]:

	0	1	2	3
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

150 rows × 4 columns

In [77]:
```
# Prepare data
#standardize
# from sklearn.preprocessing import StandardScaler
# from sklearn.model_selection import train_test_split
# scaler = StandardScaler()
# X = scaler.fit_transform(X)

#do train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

In [78]:
```
print(X_train.shape, y_train.shape)
```

```
(105, 4) (105,)
```

In [79]:
```
dt = DecisionTree(max_depth=10)


tree = dt.fit(X_train, y_train, len(set(y_train)))
pred = [dt.predict(x) for x in X_test]

print("Tree feature ind: ", tree.feature_index)
print("Tree threshold: ", tree.threshold)
print("Pred: ", np.array(pred))
print("ytest: ", y_test)
```

```
Tree feature ind:  2
Tree threshold:  2.5999999999999996
Pred:  [0 1 0 0 0 2 1 1 0 2 2 1 0 1 1 2 1 0 2 1 0 1 1 0 0 2 1 0 2 1 1 2 0 2 2 1 0
 0 1 1 0 2 1 2 2]
ytest:  [0 2 0 0 0 2 1 1 0 2 2 1 0 1 1 2 1 0 2 1 0 1 1 0 0 2 1 0 2 1 1 2 0 2 2 1 0
 0 2 1 0 2 1 2 2]
```