In [48]:

```
Name = "Muhammad Omer Farooq Bhatti"
Id = '122498'
```

In [4]:

```
import numpy as np
```

In [5]:

```
#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 [9]:

```
class dtree:
    def __init__(self, max_depth=0):
        self.max_depth=max_depth
        self.tree=[]

    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 warning about "referenced before assignments"
        feature_ix, threshold = None, None
        #print("y :" , y)

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

        # 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]
            #print("sample_sorted: ", sample_sorted)
            sort_idx = np.argsort(X[:, feature])
            y_sorted = y[sort_idx] #[0, 0, 1, 1]
            #print("y_sorted", y_sorted)

            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
                #print(sample_per_class_left)
                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_right

                # 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(y == 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,
            )
```

```
                indices_left = X[:, feature] < threshold
                X_left, y_left = X[indices_left], y[indices_left]

                #tilde for negation
                X_right, y_right = X[~indices_left], y[~indices_left]

                #take note for later decision
                node.feature_index = feature
                node.threshold = threshold
                node.left = self.fit(X_left, y_left, n_classes, depth + 1)
                node.right = self.fit(X_right, y_right, n_classes, depth + 1)
        return node

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

    def predict(self, X):
        return [self._predict(xrow) for xrow in X]

    def k_fold_cross_validation(self, X_train, y_train, depth_range, n_classes, k=10):
        #implementing k fold cross validation for determining the best value of hyperparameter k=no. of neig
        #We divide the training set into k parts. We withhold one part for testing and use the rest for trai
        #Loop through all the folds, by keeping each one separate as a testing set and using the rest for tr

        fold_size = int(X_train.shape[0]/k)
        accuracy = {}
        for d in depth_range:
            self.max_depth = d
            accuracy[self.max_depth]=[]
            for i in range(0,X_train.shape[0],fold_size):
                xtest = X_train[i:i+fold_size]
                ytest = y_train[i:i+fold_size]
                xtrain = np.concatenate((X_train[:i], X_train[i+fold_size:]), axis=0)
                ytrain = np.concatenate((y_train[:i], y_train[i+fold_size:]), axis=0)
                self.fit(xtrain, ytrain, n_classes)
                yhat = self.predict(xtest)
                accuracy[self.max_depth].append( np.sum(yhat==ytest)/ytest.shape[0] )
        return accuracy
```

In [10]:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

dataset = load_iris()
X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = train_test_split(X,y, train_size=0.7)
n_classes=len(set(y))
```

In [14]:

```
for d in range(5,16,2):
    clf = dtree(max_depth=d)
    clf.fit(X_train, y_train, n_classes)
    y_pred = clf.predict(X_test)
    accuracy= np.sum(y_pred==y_test)/y_test.shape[0]
    print(f"accuracy for max_depth = {d}: ", accuracy)
```

```
accuracy for max_depth = 5:  0.8888888888888888
accuracy for max_depth = 7:  0.9111111111111111
accuracy for max_depth = 9:  0.8888888888888888
accuracy for max_depth = 11:  0.8888888888888888
accuracy for max_depth = 13:  0.9111111111111111
accuracy for max_depth = 15:  0.8888888888888888
```

In [13]:

```
clf = dtree(max_depth=5)

d_range = [5, 7, 9, 11, 12, 13, 14, 15]
accuracy = clf.k_fold_cross_validation(X_train, y_train, d_range, n_classes, k=5 )
acc=np.zeros((len(d_range),2))
for idx, key in enumerate(d_range):
    accuracy[key] = np.mean(accuracy[key])
    print(f"Mean accuracy for k={key}: {accuracy[key]}")
    acc[idx,0] = key
    acc[idx,1] = accuracy[key]
#print(acc)
print(f"The max accuracy achieved was {acc[acc.argmax(axis=0)[1],1]} for max_depth = {acc[acc.argmax(axis=0)
```

```
Mean accuracy for k=5: 0.8952380952380953
```

```
Mean accuracy for k=7: 0.8952380952380953
Mean accuracy for k=9: 0.8857142857142858
Mean accuracy for k=11: 0.8952380952380953
Mean accuracy for k=12: 0.7238095238095239
Mean accuracy for k=13: 0.8952380952380953
Mean accuracy for k=14: 0.7333333333333334
Mean accuracy for k=15: 0.8857142857142858
The max accuracy achieved was 0.8952380952380953 for max depth = 5.0
```

In [ ]: