In [1]:
```
# Create an training dataset of an Fruit basket example
training_data = [["Green",3,"Apple"],
                 ["Red",3,"Apple"],
                 ["Red",1,"Grape"],
                 ["Blue",1,"Grape"],
                 ["Yellow",3,"Lemon"],
                 ["Green",3,"Lemon"]]
```

In [2]:
```
header  = ["Color","Size","Label"]
```

In [3]:
```
# Create a counting class that return counting different type of class / label / Output and thierr occurence
def class_count(data):
    """ Return the different type of class and thier occurence
            In Dictionary form"""
    dic = {}
    for row in data:
        i = row[-1]
        if i in dic:
            dic[i] = dic[i] + 1
        else:
            dic[i] = 1
    return dic
```

In [4]:
```
class_count(training_data)
```

```
{'Apple': 2, 'Grape': 2, 'Lemon': 2}
```

In [5]:
```
# This function return the boolean value of that the value is integer or float or not
def isnumeric(i):
    return isinstance(i,int) or isinstance(i,float)
```

In [6]:
```
isnumeric(3007)
```

```
True
```

In [7]:
```
# Create a question on the basis of dividing the data to create a tree

class Question:
    """
    Question is used to partioning the dataset

        This class just record a column number (e.g. 0 for color) and
        a column value (e.g. Green) , The 'Match' method is used to compare
        the feature value in an example to the feature value stored int the
        Question.

    """

    def __init__ (self,col,val):
        self.col = col
        self.val = val


    def match (self,example):
        """Matching the question with an example data"""
        val = example[self.col]
        if isnumeric(val):
            return (val >= self.val)
        else:
            return (val == self.val)

    # __repr__ method only for create to represent of question
    def __repr__ (self):
        condition = "=="
        if isnumeric(self.val):
            condition = ">="
        return f"Is {header[self.col]} {condition} {str(self.val)} ?"
```

Demo to check above code

In [8]:
```
Question(1,3)
```

```
Is Size >= 3 ?
```

In [9]:
```
Question (0,"Red")
```

```
Is Color == Red ?
```

In [10]:
```
q = Question(0,"Green")
q
```

```
Is Color == Green ?
```

In [11]:
```
q.match(training_data[1])
```

```
False
```

In [12]:
```
q.match(training_data[0])
```

```
True
```

Partition function is used to partition the data into two form if it is true or it is false on the basis of question and their matching function. These will return an two list first list contain rows that answer is true and second list contain that rows that answer is false. These list makes an two branch of an decision tree left side branch contain true rows and right side branch contain false rows

In [13]:
```
def partition(dataset,question):

    """These method help to partioning the dataset on the basis of question"""

    true_row,false_row = [],[]
    for row in dataset:
        if question.match(row) :
            true_row.append(row)
        else:
            false_row.append(row)
    return true_row,false_row
```

Checking the partition is working on the basis of question

In [14]:
```
q1 = Question(0,"Red")
true_row,false_row = partition(training_data,q1)
print (true_row,false_row,sep = "\n")
```

```
[['Red', 3, 'Apple'], ['Red', 1, 'Grape']]
[['Green', 3, 'Apple'], ['Blue', 1, 'Grape'], ['Yellow', 3, 'Lemon'], ['Green', 3, 'Lemon']]
```

In [15]:
```
q2 = Question(1,3)
true_row,false_row = partition(training_data,q2)
print (true_row,false_row, sep = "\n")
```

```
[['Green', 3, 'Apple'], ['Red', 3, 'Apple'], ['Yellow', 3, 'Lemon'], ['Green', 3, 'Lemon']]
[['Red', 1, 'Grape'], ['Blue', 1, 'Grape']]
```

Gini is makes an big role to partiton the ddataset on the basis of gini value.

These Gini value shows how much impurity of that dataset and calculate the information gain.

Gini Index, also known as Gini impurity, calculates the amount of probability of a specific feature that is classified incorrectly when selected randomly. If all the elements are linked with a single class then it can be called pure.

Formula :

gini = 1 - (summation i=1 --> n) (Pi**2)

Where Pi denotes the probability of an element being classified for a distinct class.

In [16]:
```
def gini(data):
    """Calculate the Gini impurity for list of rows

            ref = 'https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity'"""
    count = class_count(data)
    impurity = 1
    for row in count:
        prob_of_lbl = count[row]/float(len(data))
        impurity -= prob_of_lbl**2
    return impurity
```

Tesing the gini impurity of different type of mixing

In [17]:
```
no_mix = [["Apple"],["Apple"]]
# It gives value O because dataset with no mixing
gini(no_mix)
```

```
0.0
```

In [18]:
```
some_mixing = [["Orange"],["Apple"]]
# This will return 0.5 that means it has 50 % chances of miclassifynig
gini(some_mixing)
```

```
0.5
```

In [42]:
```
lots_of_mixing = [["Apple"],["Orange"],["Grapes"],["Grapefruit"],["Berry"]]
# it gives 0.8 becuase in this dataset there are lots of mixing
gini(lots_of_mixing)
```

```
0.7999999999999998
```

Information Gain is applied to quantify which feature provides maximal information about the classification based on the notion of entropy, i.e. by quantifying the size of uncertainty, disorder or impurity, in general, with the intention of decreasing the amount of entropy initiating from the top (root node) to bottom(leaves nodes).

In [43]:
```
def info_gain(left,right,current_uncertainity):
    """Informatin Gain
            The uncertainity of the starting node , minus the weight impurity of
            two child
    """
    prob = float(len(left))/(len(left) + len(right))
    return current_uncertainity - prob*gini(left) - (1-prob)*gini(right)
```

Demo of what information gain after partitioning the dataset

In [21]:
```
# Calculate the uncertainity of current data
current_uncertainity = gini (training_data)
current_uncertainity
```

```
0.6666666666666665
```

In [22]:
```
# How much information do we gain by partioning by "Green"
true_row,false_row = partition(training_data,Question(0,"Green"))
info_gain(true_row,false_row,current_uncertainity)
```

```
0.08333333333333315
```

In [44]:
```
# How much information do we gain by partioning by "Red"
true_row,false_row = partition(training_data,Question(0,"Red"))
info_gain(true_row,false_row,current_uncertainity)
```

```
0.08333333333333315
```

Below function is used to finding best question to partition the dataset on the basis of best information gain

In [24]:
```
def find_best_split(data):

    """find the best question to ask by iterating over every feature / value
    and calculating information gain"""
    best_gain = 0 # keep track of best information gain
    best_question = None # keep track of feature / value produced it
    feature = len(data[0]) - 1 # number of columns
    current_uncertainity = gini(data)
    for col in range(feature): # for each feature
        value = set([row[col] for row in data]) # unique value of columns
        for val in value: # for each value
            question = Question(col,val) # Evaluate question on the basis of column and value
            true_row,false_row = partition(data,question) # partitioning the data on the basis of
 question
            if (len(true_row) == 0) or (len(false_row) == 0): # skip the result if doesn't divide
 the dataset
                continue
            gain = info_gain(true_row,false_row,current_uncertainity) # Evaluating the information
gain
            if (gain > best_gain): # Updating the information gain and question
                best_gain = gain
                best_question = question
    return best_gain, best_question
```

In [25]:
```
find_best_split(training_data)
```

```
(0.3333333333333332, Is Size >= 3 ?)
```

In [26]:
```
class Leaf:
    """A leaf node classifie data
    This hold a dictionary of class (e.g. Apple) number of times
    it appears in the row from the training data that reach this leaf"""
    def __init__(self,rows):
        self.prediction = class_count(rows)
```

In [27]:
```
class Decision_Node:
    """A decision Node asks a question

    This hold a reference to the question and to the child nodes"""

    def __init__(self,question,true_branch,false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch
```

In [28]:
```
def build_tree(rows):
    """Builds the tree.

    Rules of recursion : 1) Believe that it works, 2) Start by checking
    for the base case (no further information gain), 3) Prepare for
    giant stack traces"""

    # try partitioning the dataset on each of the unique attributr
    # Calculate the information gain
    # and return the question that produce the highest gain
    gain,question = find_best_split(rows)

    # base case : no further inforamation gain
    # Since we can ask no further question
    # we'll return a leaf
    if (gain == 0):
        return Leaf(rows)

    # if we reach here we found a useful feature / value
    # to partion on
    true_row,false_row = partition(rows,question)

    # recursively build the true branch
    true_branch = build_tree(true_row)

    # recursively build the false branch
    false_branch = build_tree(false_row)

    # return a question
    # This records the best feature / value to ask at this point
    # as well as the branches to follow
    # depending on the answer
    return Decision_Node(question, true_branch, false_branch)
```

In [29]:
```
def print_tree(node , spacing = "  "):

    # base case we've reached a leaf
    if isinstance(node,Leaf):
        print (spacing+ "Predict", node.prediction)
        return

    # print the question at this node
    print (spacing + str(node.question))

    # call this function recursively on the true branch
    print (spacing + "--> True : ")
    print_tree(node.true_branch, spacing + " ")

    # call this function recursively on the false branch
    print (spacing+"--> False : ")
    print_tree (node.false_branch, spacing + "  ")
```

In [30]:
```
my_tree = build_tree(training_data)
```

In [31]:
```
print_tree(my_tree)
```

```
Is Size >= 3 ?
--> True :
 Is Color == Red ?
 --> True :
  Predict {'Apple': 1}
 --> False :
  Is Color == Green ?
  --> True :
   Predict {'Apple': 1, 'Lemon': 1}
  --> False :
   Predict {'Lemon': 1}
--> False :
  Predict {'Grape': 2}
```

In [32]:
```
def classify(row, node):
    # Base cases we've reached at leaf
    if (isinstance(node,Leaf)):
        return node.prediction

    # Decide whether to follow the true branch or the false branch
    # Compare the feature / value stored in the node
    # to the example we'er consider
    if node.question.match(row):
        return classify(row,node.true_branch)
    else:
        return classify(row,node.false_branch)
```

In [33]:
```
# Tree predict the first row of our
# training data is an apple with confidence 1
classify(training_data[0],my_tree)
```

```
{'Apple': 1, 'Lemon': 1}
```

In [34]:
```
def print_leaf(counts):
    """A nicer way to print the prediction at a leaf"""
    total = sum(counts.values())*1.0
    prob = {}
    for x in counts.keys():
        prob[x] = str(int(counts[x]/total*100)) + '%'
    return prob
```

In [35]:
```
print_leaf(classify(training_data[0],my_tree))
```

```
{'Apple': '50%', 'Lemon': '50%'}
```

In [36]:
```
print_leaf(classify(["Green",3],my_tree))
```

```
{'Apple': '50%', 'Lemon': '50%'}
```

In [37]:
```
# Final testing
testing_data = [["Green",8,"Watermelon"],
                ["Yellow",4,"Apple"],
                ["Green",3,"Apple"],
                ["Red",2,"Grape"],
                ["Red",1,"Grape"],
                ["Yellow",3,"Lemon"]]
```

In [41]:
```
for row in testing_data:
    print (" Actual : %s, Predicted : %s" % (row[-1],print_leaf(classify(row,my_tree))))
```

```
 Actual : Watermelon, Predicted : {'Apple': '50%', 'Lemon': '50%'}
 Actual : Apple, Predicted : {'Lemon': '100%'}
 Actual : Apple, Predicted : {'Apple': '50%', 'Lemon': '50%'}
 Actual : Grape, Predicted : {'Grape': '100%'}
 Actual : Grape, Predicted : {'Grape': '100%'}
 Actual : Lemon, Predicted : {'Lemon': '100%'}
```

In [ ]: