Decision Tree Algorithm

Using a Toy Dataset (Fruits)

Intuition

The decision tree algorithm is a type of supervised learning which can be used for solving both classification and regression problems. In this examle we will solve classification problems. Therefore, our algorithm is called a categorical variable decision tree. Essentially, the algorithm predicts a class label for a record from the root of the 'tree'. It compares the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

Steps

- 1.) Split the data into training and testing sets
- 2.) Find the most relevant question
- 3.) Add/Build a decision tree
- 4.) Go to step 1 until arrived at the answer

Set up (Libraries Used)

In [308]:
```
1 import pandas as pd
2 from sklearn.model_selection import train_test_split
```

Step 1.) Create the Toy Training and Testing Dataset

In [286]:

```
# Training Dataset
training = [
    ['Green', 3, 'Apple'],
    ['Yellow', 3, 'Apple'],
    ['Red', 1, 'Grape'],
    ['Red', 1, 'Grape'],
    ['Yellow', 3, 'Lemon'],
]

# Testing Dataset
testing = [
    ['Green', 3, 'Apple'],
    ['Yellow', 4, 'Apple'],
    ['Red', 2, 'Grape'],
    ['Red', 1, 'Grape'],
    ['Yellow', 3, 'Lemon'],
]

# Header names
header = ["color", "diameter", "label"]
```

Step 2.) Find the Best Question and Partition

Define functions that will help find the best partition. To find the best partition means to find the best question to split the data.

In [316]:

```
# Counts the number of each type of example in a dataset
def class_counts(rows):
    counts = {}
    for row in rows:
        label = row[-1]
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts

# Test if a value is numeric
def is_numeric(value):
    return isinstance(value, int) or isinstance(value, float)

# Define the question
class Question:
    def __init__(self, column, value):
        self.column = column
        self.value = value

    def match(self, example):

        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (
            header[self.column], condition, str(self.value))

# SPlit between True rows and False rows based on the question
def partition(rows, question):
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows

# Calculate the Gini Impurity
def gini(rows):
    counts = class_counts(rows)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(rows))
        impurity -= prob_of_lbl**2
    return impurity

# Calculate the info gained
def info_gain(left, right, current_uncertainty):
```

```
    p = float(len(left)) / (len(left) + len(right))
    return current_uncertainty - p * gini(left) - (1 - p) * gini(right)


```

Determine Current Uncertainty

In [287]:
```
current_uncertainty = gini(training)
current_uncertainty
```

Out[287]:
```
0.6399999999999999
```

Partition the data

In [317]:
```
def find_best_split(rows):
    """Find the best question to ask by iterating over every feature /
    and calculating the information gain."""
    best_gain = 0  # keep track of the best information gain
    best_question = None  # keep train of the feature / value that proc
    current_uncertainty = gini(rows)
    n_features = len(rows[0]) - 1  # number of columns

    for col in range(n_features):  # for each feature

        values = set([row[col] for row in rows])  # unique values in th

        for val in values:  # for each value

            question = Question(col, val)

            # try splitting the dataset
            true_rows, false_rows = partition(rows, question)

            # Skip this split if it doesn't divide the
            # dataset.
            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = info_gain(true_rows, false_rows, current_uncertainty

            # You actually can use '>' instead of '>=' here
            # but I wanted the tree to look a certain way for our
            # toy dataset.
            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question
```

Results -

In [321]:

```
best_gain, best_question = find_best_split(training_data)
print("Best Question : ",best_question)
print("The gain from the best question : ", best_gain )
```

```
Best Question :  Is diameter >= 3?
The gain from the best question :  0.37333333333333324
```

Step 3.) Build a Decision Tree

In [325]:

```
# LEaf node classifies the data
class Leaf:
    def __init__(self, rows):
        self.predictions = class_counts(rows)

# Asks a question and holds refrences to child nodes
class Decision_Node:
    def __init__(self,
                 question,
                 true_branch,
                 false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch

# Builds the tree
def build_tree(rows):
    gain, question = find_best_split(rows)

    if gain == 0:
        return Leaf(rows)

    true_rows, false_rows = partition(rows, question)

    true_branch = build_tree(true_rows)

    false_branch = build_tree(false_rows)

    return Decision_Node(question, true_branch, false_branch)

# To Print the tree
def print_tree(node, spacing=""):

    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    print (spacing + str(node.question))

    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + "   ")

    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + "   ")
```

In [326]:
```
my_tree = build_tree(training)
print_tree(my_tree)
```

```
Is diameter >= 3?
--> True:
  Is color == Yellow?
  --> True:
    Predict {'Apple': 1, 'Lemon': 1}
  --> False:
    Predict {'Apple': 1}
--> False:
  Predict {'Grape': 2}
```

Classification

Functions to classify and print the leaf(predictions)

In [327]:
```
def classify(row, node):
    if isinstance(node, Leaf):
        return node.predictions

    if node.question.match(row):
        return classify(row, node.true_branch)
    else:
        return classify(row, node.false_branch)

def print_leaf(counts):
    total = sum(counts.values()) * 1.0
    probs = {}
    for lbl in counts.keys():
        probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
    return probs
```

In [302]:
```
print_leaf(classify(training_data[0], my_tree))
```

Out[302]:
```
{'Apple': '100%'}
```

Test the Classifier on the Test data

In [304]:
```
for row in testing:
    print ("Actual: %s. Predicted: %s" %
           (row[-1], print_leaf(classify(row, my_tree))))
```

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

In [ ]:
```

```