Worksheet 13

Name: Maha Alali UID: U84088912

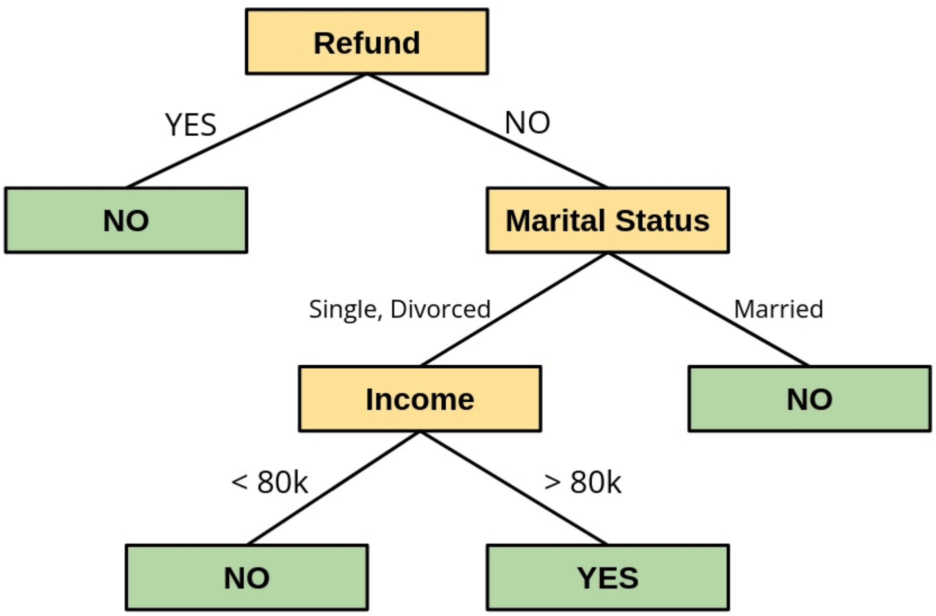
Topics

- Decision Trees

Decision Trees

In [24]:
```
from IPython.display import Image
Image(filename="tree.jpg", width=500, height=300)
```

Out[24]:



Using the above Decision Tree, what class would you predict for the following unseen record:

Refund	Marital Status	Income
No	Married	90k

class = NO

Working with a dataset that attempts to understand the relationship between `heart disease` and whether or not a person experiences `chest_pain` and/or has `thalassemia` . All the attributes are binary (either 0 or 1) for simplicity.

In [25]:
```
import numpy as np

data = np.genfromtxt(fname='./dataset.tsv', delimiter = '\t', names = Tr
```

a) Before splitting the dataset at all, we observe the following distribution of 1s and 0s in the `heart_disease` class:

In [26]:
```
print(data["heart_disease"])
```

```
[1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0.
0.
 0. 1. 1. 1.]
```

write a function that calculates the GINI of that node.

In [27]:
```
def gini(node):
    if(len(node) == 0):
        return 0
    frequencies = [
        sum(node)/len(node),
        1-sum(node)/len(node)
    ]
    return 1 - sum([f**2 for f in frequencies])

print("GINI of the node is ", gini(data["heart_disease"]))
```

```
GINI of the node is  0.48979591836734704
```

b) Write a function that computes the gini of a split.

In [28]:
```
def gini_split(data, attr, target_name):
    data[data[attr] == 0]
    data[data[attr] == 1]


    subsets = [
        data[data[attr] == 0][target_name],
        data[data[attr] == 1][target_name]
    ]

    return sum([gini(x)* len(x) for x in subsets])/len(data)

print("GINI of split on thalassemia = ", gini_split(data, "thalassemia",
print("GINI of split on chest_pain = ", gini_split(data, "chest_pain", "
```

```
GINI of split on thalassemia =  0.23469387755102047
GINI of split on chest_pain =  0.4419642857142857
```

We can represent a decision tree recursively with the `Node` class below.

In [29]:
```
class Node:
    def __init__(self, attribute):
        self.attr = attribute
        self.left = None
        self.right = None
        self.vote = None

    def _node_at(self, depth):
        pretty_print = ""
        if self.left is not None:
            for _ in range(depth):
                pretty_print += "| "
            pretty_print += self.attr + ' = 0: \n'
            pretty_print += self.left._node_at(depth + 1)

        if self.right is not None:
            for _ in range(depth):
                pretty_print += "| "
            pretty_print += self.attr + ' = 1: \n'
            pretty_print += self.right._node_at(depth + 1)

        if self.right is None and self.left is None:
            for _ in range(depth):
                pretty_print += "| "
            pretty_print += "vote = " + str(self.vote) + '\n'

        return pretty_print

    def __repr__(self):
        return self._node_at(0)

B = Node("B")
C = Node("C")
left_leaf = Node("leaf")
left_leaf.vote = 0
right_leaf = Node("leaf")
right_leaf.vote = 1

B.right = right_leaf
B.left = left_leaf
C.right = right_leaf
C.left = left_leaf

tree = Node("A")
tree.left = B
tree.right = C

print(tree)
```

```
A = 0:
| B = 0:
| | vote = 0
| B = 1:
| | vote = 1
A = 1:
| C = 0:
| | vote = 0
| C = 1:
| | vote = 1
```

Each node is defined by splitting the dataset on a specific attribute. If the attribute value is 0, we explore the left node, if the attribute value is 1, we explore the right node. The left and right nodes are both of type `Node` . If the node has no left node and no right node then it is a leaf node and should contain a vote for what class should be predicted.

c) Write a function that takes in a decision tree and a data point, and walks through the tree based on the data point's attribute values to predict its class.

Type *Markdown* and LaTeX: α^2

In [30]:
```
def predict(tree : Node, example):
    if tree.left is None and tree.right is None:
        return tree.vote

    if example[tree.attr] == 0:
        return predict(tree.left, example)

    if example[tree.attr] == 1:
        return predict(tree.right, example)

    return 0

print(predict(tree, {"A": 0, "B": 1, "C": 0})) # A -> B -> right
print(predict(tree, {"A": 0, "B": 0, "C": 0})) # A -> B -> left
print(predict(tree, {"A": 1, "B": 1, "C": 0})) # A -> C -> left
print(predict(tree, {"A": 1, "B": 1, "C": 1})) # A -> C -> right
```

```
1
0
0
1
```

d) Write a function that finds the best attribute to split on wrt the GINI of the split. Recall a smaller GINI is better.

In [31]:
```
def get_best_attribute(data, target_name):
    best_attr = None
    min_gini = float('inf')
    for attr in data.dtype.names:
        if attr != target_name:
            gini = gini_split(data, attr, target_name)

            if gini < min_gini:
                min_gini = gini
                best_attr = attr
    return best_attr
```

e) Complete the code below to build a `SimpleDecisionTree` on the dataset provided.

In [32]:
```
class SimpleDecisionTree:

    def __init__(self, max_depth, data, target_name):
        self.max_depth = max_depth
        self.data = data
        self.target_name = target_name
        self.tree = None
        self.default_class = None


    def __repr__(self):
        return self.tree.__repr__()


    def get_subset(self, data, attr):
        subset_1 = data[data[attr] == 0]
        subset_2 = data[data[attr] == 1]
        return subset_1, subset_2


    def gini_split(self, data, attr):
        subsets = [
            data[data[attr] == 0][self.target_name],
            data[data[attr] == 1][self.target_name]
        ]

        return sum([gini(x)* len(x) for x in subsets])/len(data)


    def get_majority_vote(self, data):
        # Initialize a dictionary to count occurrences
        counts = {}

        # Iterate through each item in the target column
        for value in data[self.target_name]:
            if value in counts:
                counts[value] += 1
            else:
                counts[value] = 1

        # Find the key with the maximum value (majority vote)
        majority_vote = max(counts, key=counts.get)

        return majority_vote


    def get_best_attribute(self, data):
        best_attr = None
        min_gini = float('inf')
        for attr in data.dtype.names:
            if attr != self.target_name:
                gini = gini_split(data, attr, self.target_name)

                if gini < min_gini:
                    min_gini = gini
                    best_attr = attr
        return best_attr
```

```python
    def build_tree(self, data, depth):
        attr = self.get_best_attribute(data)
        node = Node(attr)

        if depth == 0:
            if data is None:
                node.vote = self.default_class
            else:
                node.vote = self.get_majority_vote(data)
            return node

        left, right = self.get_subset(data, node.attr)

        node.left = self.build_tree(left, depth - 1)
        node.right = self.build_tree(right, depth - 1)

        if node.left is None and node.right is None:
            node.vote = self.get_majority_vote(data)

        return node


    def train(self):
        if self.max_depth > len(self.data.dtype.names) - 1:
            self.max_depth = len(self.data.dtype.names) - 1

        self.default_class = self.get_majority_vote(self.data)
        self.tree = self.build_tree(self.data, self.max_depth)


simple_tree = SimpleDecisionTree(2, data, "heart_disease")
simple_tree.train()
print(simple_tree)
```

```
thalassemia = 0:
| chest_pain = 0:
| | vote = 0.0
| chest_pain = 1:
| | vote = 0.0
thalassemia = 1:
| chest_pain = 0:
| | vote = 1.0
| chest_pain = 1:
| | vote = 1.0
```