Assignment 1

Name:

Roll Number:

In [1]:
```
# import all the necessary libraries here
import pandas as pd
from sklearn import model_selection
```

In [2]:
```
df = pd.read_csv('../../dataset/decision-tree.csv')
print(df.shape)
```

```
(768, 9)
```

In [3]:
```
import math

def unique_vals(rows, col):
    """Find the unique values for a column in a dataset."""
    return set([row[col] for row in rows])

#######
# Demo:
# unique_vals(training_data, 0)
# unique_vals(training_data, 1)
#######


def class_counts(rows):
    """Counts the number of each type of example in a dataset."""
    counts = {}  # a dictionary of label -> count.
    for row in rows:
        # in our dataset format, the label is always the last column
        label = row[-1]
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts

#######
# Demo:
# class_counts(training_data)
#######




def max_label(dict):
    max_count = 0
    label = ""

    for key, value in dict.items():
        if dict[key] > max_count:
            max_count = dict[key]
            label = key

    return label




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

#######
# Demo:
# is_numeric(7)
# is_numeric("Red")
#######
```

```python
class Question:
    """A Question is used to partition a dataset.

    This class just records 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 in the
    question. See the demo below.
    """

    def __init__(self, column, value, header):
        self.column = column
        self.value = value
        self.header = header

    def match(self, example):
        # Compare the feature value in an example to the
        # feature value in this question.
        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        # This is just a helper method to print
        # the question in a readable format.
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (
            self.header[self.column], condition, str(self.value))


def partition(rows, question):
    """Partitions a dataset.

    For each row in the dataset, check if it matches the question. If
    so, add it to 'true rows', otherwise, add it to 'false rows'.
    """
    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


def gini(rows):
    """Calculate the Gini Impurity for a list of rows.

    There are a few different ways to do this, I thought this one was
    the most concise. See:
    https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity
    """
    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

## TODO: Step 3
def entropy(rows):

    # compute the entropy.
    entries = class_counts(rows)
    avg_entropy = 0
    size = float(len(rows))
    for label in entries:
        prob = entries[label] / size
        avg_entropy = avg_entropy + (prob * math.log(prob, 2))
    return -1*avg_entropy


def info_gain(left, right, current_uncertainty):
    """Information Gain.

    The uncertainty of the starting node, minus the weighted impurity of
    two child nodes.
    """
    p = float(len(left)) / (len(left) + len(right))

    ## TODO: Step 3, Use Entropy in place of Gini
    return current_uncertainty - p * entropy(left) - (1 - p) * entropy(right)

def find_best_split(rows, header):
    """Find the best question to ask by iterating over every feature / value
    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 produced it
    current_uncertainty = entropy(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 the colum

        for val in values:  # for each value

            question = Question(col, val, header)

            # 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

## TODO: Step 2
class Leaf:
    """A Leaf node classifies data.

    This holds a dictionary of class (e.g., "Apple") -> number of times
    it appears in the rows from the training data that reach this leaf.
    """

    def __init__(self, rows, id, depth):
        self.predictions = class_counts(rows)
        self.predicted_label = max_label(self.predictions)
        self.id = id
        self.depth = depth

## TODO: Step 1
class Decision_Node:
    """A Decision Node asks a question.

    This holds a reference to the question, and to the two child nodes.
    """

    def __init__(self,
                 question,
                 true_branch,
                 false_branch,
                 depth,
                 id,
                 rows):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch
        self.depth = depth
        self.id = id
        self.rows = rows


## TODO: Step 3
def build_tree(rows, header, depth=0, id=0):
    """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.
    """
    # depth = 0
    # Try partitioing the dataset on each of the unique attribute,
    # calculate the information gain,
    # and return the question that produces the highest gain.

    if(depth>10):
```

```
        return Leaf(rows,id,depth)
    gain, question = find_best_split(rows, header)

    # Base case: no further info gain
    # Since we can ask no further questions,
    # we'll return a leaf.
    if gain == 0:
        return Leaf(rows, id, depth)

    # If we reach here, we have found a useful feature / value
    # to partition on.
    # nodeLst.append(id)
    true_rows, false_rows = partition(rows, question)

    # Recursively build the true branch.
    true_branch = build_tree(true_rows, header, depth + 1, 2 * id + 2)

    # Recursively build the false branch.
    false_branch = build_tree(false_rows, header, depth + 1, 2 * id + 1)

    # Return a Question node.
    # This records the best feature / value to ask at this point,
    # as well as the branches to follow
    # depending on on the answer.
    return Decision_Node(question, true_branch, false_branch, depth, id, rows)

## TODO: Step 8 - already done for you
def prune_tree(node, prunedList):
    """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.
    """

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        return node
    # If we reach a pruned node, make that node a leaf node and return. Since
    # below it are automatically not considered
    if int(node.id) in prunedList:
        return Leaf(node.rows, node.id, node.depth)

    # Call this function recursively on the true branch
    node.true_branch = prune_tree(node.true_branch, prunedList)

    # Call this function recursively on the false branch
    node.false_branch = prune_tree(node.false_branch, prunedList)

    return node

## TODO: Step 6
def classify(row, node):
    """See the 'rules of recursion' above."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
```

```
        return node.predicted_label

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

## TODO: Step 4
def print_tree(node, spacing=""):
    """World's most elegant tree printing function."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        print(spacing + "Leaf id: " + str(node.id) + " Predictions: " + str(nc
        return

    # Print the question at this node
    print(spacing + str(node.question) + " id: " + str(node.id) + " depth: " +

    # 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 + "  ")


def print_leaf(counts):
    """A nicer way to print the predictions at a leaf."""
    total = sum(counts.values()) * 1.0
    probs = {}
    for lbl in counts.keys():
        probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
    return probs

## TODO: Step 5
def getLeafNodes(node, leafNodes =[]):

    # Base case
    if isinstance(node, Leaf):
        leafNodes.append(node)
        return

    # Recursive right call for true values
    getLeafNodes(node.true_branch, leafNodes)

    # Recursive left call for false values
    getLeafNodes(node.false_branch, leafNodes)

    return leafNodes


def getInnerNodes(node, innerNodes =[]):
```

```
    # Base case
    if isinstance(node, Leaf):
        return

    innerNodes.append(node)

    # Recursive right call for true values
    getInnerNodes(node.true_branch, innerNodes)

    # Recursive left call for false values
    getInnerNodes(node.false_branch, innerNodes)

    return innerNodes

## TODO: Step 6
def computeAccuracy(rows, node):

    count = len(rows)
    if count == 0:
        return 0

    accuracy = 0
    for row in rows:
        # last entry of the column is the actual label
        if row[-1] == classify(row, node):
            accuracy += 1
    return round(accuracy/count, 2)

# default data set

header = list(df.columns)

# overwrite your data set here
# header = ['SepalL', 'SepalW', 'PetalL', 'PetalW', 'Class']
# df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/
# data-set link: https://archive.ics.uci.edu/ml/machine-learning-databases/bre
# df = pd.read_csv('data_set/breast-cancer.csv')


lst = df.values.tolist()

# splitting the data set into train and test
trainDF, testDF = model_selection.train_test_split(lst, test_size=0.2)

# building the tree
t = build_tree(trainDF, header)

# get leaf and inner nodes
print("\nLeaf nodes ****************")
leaves = getLeafNodes(t)
for leaf in leaves:
    print("id = " + str(leaf.id) + " depth =" + str(leaf.depth))

print("\nNon-leaf nodes ****************")
innerNodes = getInnerNodes(t)
```

```
for inner in innerNodes:
    print("id = " + str(inner.id) + " depth =" + str(inner.depth))

# print tree
maxAccuracy = computeAccuracy(testDF, t)
print("\nTree before pruning with accuracy: " + str(maxAccuracy*100) + "\n")
print_tree(t)

# TODO: You have to decide on a pruning strategy
# Pruning strategy
nodeIdToPrune = -1
import copy
t1=copy.deepcopy(t)
for node in innerNodes:
    if node.id != 0:
        prune_tree(t, [node.id])
        currentAccuracy = computeAccuracy(testDF, t)
        print("Pruned node_id: " + str(node.id) + " to achieve accuracy: " + s
        # print("Pruned Tree")
        # print_tree(t)
        if currentAccuracy > maxAccuracy:
            maxAccuracy = currentAccuracy
            nodeIdToPrune = node.id
        t = t1
        if maxAccuracy == 1:
            break

if nodeIdToPrune != -1:
    t = build_tree(trainDF, header)
    prune_tree(t, [nodeIdToPrune])
    print("\nFinal node Id to prune (for max accuracy): " + str(nodeIdToPrune)
else:
    t = build_tree(trainDF, header)
    print("\nPruning strategy did'nt increased accuracy")

print("\n*******************************************************************"
print("*********** Final Tree with accuracy: " + str(maxAccuracy*100) + "%  **
print("*******************************************************************\n"
print_tree(t)





import graphviz

# Convert the decision tree into a format suitable for Graphviz
def build_graphviz_tree(node, dot=None):
    if dot is None:
        dot = graphviz.Digraph(format='png')

    if isinstance(node, Leaf):
        label = f"Predicted: {node.predicted_label}\n{print_leaf(node.predicti
        dot.node(str(node.id), label, shape='box')
    else:
        dot.node(str(node.id), str(node.question))

        # True branch
```

```
        if isinstance(node.true_branch, Leaf):
            label = f"Predicted: {node.true_branch.predicted_label}\n{print_le
            dot.node(str(node.true_branch.id), label, shape='box')
            dot.edge(str(node.id), str(node.true_branch.id), label='True')
        else:
            dot.node(str(node.true_branch.id), str(node.true_branch.question))
            dot.edge(str(node.id), str(node.true_branch.id), label='True')
            build_graphviz_tree(node.true_branch, dot)

        # False branch
        if isinstance(node.false_branch, Leaf):
            label = f"Predicted: {node.false_branch.predicted_label}\n{print_l
            dot.node(str(node.false_branch.id), label, shape='box')
            dot.edge(str(node.id), str(node.false_branch.id), label='False')
        else:
            dot.node(str(node.false_branch.id), str(node.false_branch.questior
            dot.edge(str(node.id), str(node.false_branch.id), label='False')
            build_graphviz_tree(node.false_branch, dot)

    return dot

# Build the Graphviz tree using the Decision Node structure
graphviz_tree = build_graphviz_tree(t)

# Save the Graphviz tree as an image
graphviz_tree.render('new1', cleanup=True)

print("Decision tree visualization saved as 'new1.png'")
```

```
Leaf nodes ****************
id = 30 depth =4
id = 122 depth =6
id = 121 depth =6
id = 120 depth =6
id = 119 depth =6
id = 478 depth =8
id = 956 depth =9
id = 1912 depth =10
id = 1911 depth =10
id = 476 depth =8
id = 1906 depth =10
id = 3812 depth =11
id = 3811 depth =11
id = 1904 depth =10
id = 3808 depth =11
id = 3807 depth =11
id = 117 depth =6
```

In [4]:

```
graphviz_tree.view()
```

Out[4]: 'new1.png'

In [ ]:

```

```