Assignment 1: Decision Tree

Instructions

- There are two parts to this assignment. The first part requires you to solve some theoretical/numerical questions, and the second part requires you to code a decision tree in Python.
- The solution to the first part can be typed or handwritten legibly.
- For the programming part, please use parameters and not hard coded paths or values. All instructions for compiling and running your code must be placed in the README file.
- All work submitted must be your own. Do not copy from online sources. If you use any references, please list them.
- You should use a cover sheet which can be downloaded from this link
- You are allowed to work in pairs i.e. a group of two students is allowed. Please write the names of the group members on the cover page.
- **You have a total of 4 free late days for the entire semester. You can use at most 2 days for any one assignment. After four days have been used up, there will be a penalty of 10% for each late day. The submission for this assignment will be closed 2 days after the due date.**
- Please ask all questions on Piazza, not via email.

Part I: Written Problems (30 points)

1. Representing Boolean Functions (10 points)

Give decision trees to represent the following concepts. Your decision tree must contain as few nodes as possible. You can assume A, B, and C are Boolean variables.

(a) $Y = (\neg A \vee B) \wedge \neg(C \wedge A)$ where ¬ represents the NOT operator

(b) $Y = (A \oplus B) \wedge C$ where the symbol $\oplus$ represents the XOR logical operator

(c) $Y = (A \vee B) \wedge (B \vee C) \wedge (A \vee C)$

(d) $Y = (A \vee B) \wedge \neg A \wedge \neg B$ where ¬ represents the NOT operator

2. Decision Trees (20 points)

In this question, you will use the ID3 algorithm to create a decision tree for the dataset given below. There are three Boolean attributes X1, X2, and X3 and a Boolean class attribute.
Be sure to show detailed calculations for each step including entropy and information gain values. Draw a plot of the final tree that you obtain and show the class labels for the leaf nodes. Also indicate the set of instances that are associated with each leaf node.

Instance	X1	X2	X3	Class
1	1	0	0	1
2	0	1	0	1
3	0	0	0	0
4	1	0	1	0
5	0	0	0	0
6	1	1	0	1
7	0	1	1	0
8	1	0	0	1
9	0	0	0	0
10	1	0	0	1

Programming Part (70 points)

In this part, you will code a decision tree in Python without using any machine learning/data mining libraries. You are free to use data loading and parsing libraries, such as NumPy or Pandas. As a starting point, we will use the code written by **Google Developers**, which is available at `https://github.com/random-forests/tutorials/blob/master/decision_tree.py`. There is a video explaining this code available at `https://www.youtube.com/watch?v=LDRbO9a6XPU`. This will give you a good starting point for moving forward into more complex things. We have provided you two files - *DecisionTree.py* and *driver.py* that contain areas that are marked **TODO**. Be sure to complete them in the order mentioned below.

Below are the steps that you need to do in proper order. Remember that the next step depends on previous, so be sure to complete each step before moving to the next one.

1. The code has the following dependencies, which need to be installed before running this code:

   (a) Pandas. More details at: https://pandas.pydata.org/

   (b) Scikit Learn for only one method in the driver code - train test split

   You have to make changes only in the *DecisionTree.py* file to get everything in the *driver.py* to run.

2. The code for each *decision node* in class **Decision_Node** stores information about non-leaf nodes. Besides the information given in the code on line 278 of the original code, you need to add following properties for each node:

   - depth of the node - starting from the root node, which is at depth 0
   - id of the node, using the following schema:
     The root node will have an id of 0. We assume that each node will split into two branches only - left and right. If a node has id of n, then its left child will have id of $2n + 1$ and right child will have id of $2n + 2$. This will ensure that there is no clash in node ids.
   - the rows of data that it contains

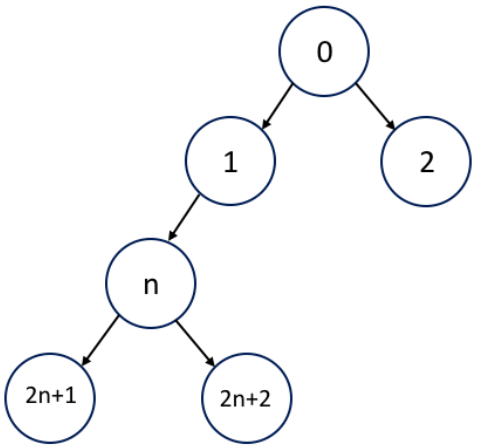

Figure 1: Numbering of Nodes in the Decision Tree

3. We will modify the definition of the leaf node in line 267 so that it stores the following properties besides the ones given:

   - predicted class label. Note that the **predictions** is a dictionary that contains the class counts. You will need to extract the majority class in it and output that.
   - id of the leaf node
   - depth of the leaf node

4. Next, you need to modify the **build_tree** function. This is the main function that builds the tree from top (root mode) to bottom (leaf nodes) recursively. You need to modify the tree definition based on the conditions of parts 1 and 2 above. To construct each node, you need to pass besides the **rows**, other parameters such as *depth, id, and header information.* The header information is necessary because we will not hard code this information, like it is done in line 27 of the original code.

   The new definition of the **build_tree** function would be as follows:

```
def build_tree(rows, depth, id, header):
...
 if gain == 0:
        return Leaf(rows, id, depth)

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

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

 # 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)
```

Make sure you finish and test this function before going ahead. Also, the example finds the best split using the **find_best_split** function that internally uses the function **info_gain** function, which uses the Gini index for impurity (line 138 of original code). In class, we have studied Entropy measure. You would need to replace the **gini** function with an **entropy** function.

5. Now, we will modify the **print_tree** function to print more details from the tree. Besides the current details that are outputted, we would also like to output the depth, node id of the decision nodes. For the leaf label, we need to see the **predicted class label**. You can output the predicted dictionary also, but that is optional.

6. Next, you will write two functions - one that returns a list of all inner nodes (i.e. those nodes that are not leaf nodes) and another function that returns a list of leaf nodes. Note that you have to return the nodes, not just their ids. You can call these functions **getInnerNodes** and **getLeafNodes**. Both of these should have one input parameter - the root node of the tree. You can see an example of this in the **print_tree** function - without the spacing parameter, of course.

7. We will modify the **classify** function so that it outputs just the predicted class label, instead of class counts. The **classify** function works on any 1 row of data. We will also create a function called **computeAccuracy** that takes in a set of rows and returns their accuracy.
   Below is the signature of that function:

```
def computeAccuracy(rows, node):
...
 totalRows = len(rows)
 numAccurate = 0
 for row in rows:
  call the classify function to get predicted label.
  If predicted label = true label:
   numAccurate = numAccurate + 1
...

 return round(numAccurate/totalRows, 2)
```

8. For testing our model, we will use Scikit Learn's train_test_split method. This will allow you to choose a random sample of the dataset for training and remaining for test.

9. **Post-Pruning** the tree: The pruning algorithm can be understood by reading the section 3.7.1.1 (Reduced Error Pruning) in the textbook. Below is an illustration for this: Suppose we would like to prune node 2, then we would remove the sub-tree rooted at 2 i.e. remove everything below 2 and make it the leaf node. The prediction will be obtained by taking the majority label at the node 2.

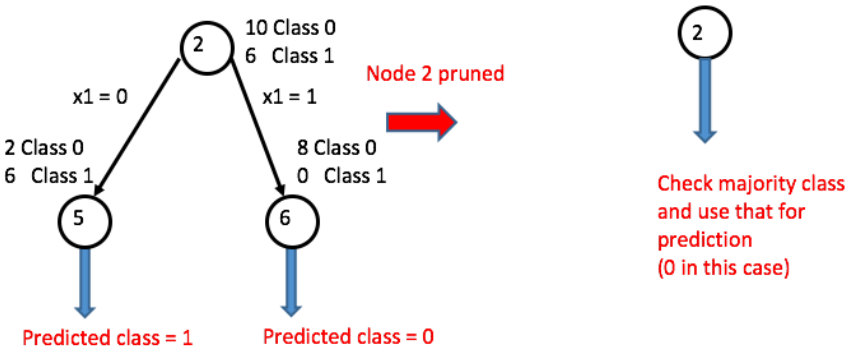

Figure 2: Post-Pruning the Decision Tree

To help you with this, I have already implemented the *prune_tree* function which takes in two parameters:

(a) Root node of the tree that you would like to prune

(b) The ids of the node that you want to prune as a list

It returns the root node of the pruned tree.

The program uses recursion and goes through the nodes of the tree, if it finds a node that needs to be pruned, it is made a leaf node. This way, the sub-tree below that node will not be needed. If you want to prune

a node and a node in the sub-tree below it, for example, 2 and 5 above, then pruning 2 will already prune 5 and there is no need to do anything else.

You would need to decide a **pruning strategy**, which could be any of the following or your own:

(a) Prune each node that is 1 level above the leaf e.g. node 2 in the figure above and see if it helps improve the test error

(b) Randomly select n nodes and prune them as a batch, and see if it improves accuracy

(c) Anything else that you can think of

10. After making the above changes, you have to get everything in the *driver.py* file to run without error. and then choose another dataset from UCI Machine Learning dataset that is suitable for classification tasks and get it to work using the code that you created earlier.

What to submit:
You need to submit the following for the programming part:

- Your source code including the driver code file
- Output for the datasets before and after pruning using **your pruning strategy**
- A brief report summarizing your results and details of your pruning strategy
- Any assumptions that you made or any bugs in the supplied code that you found.