CS311 – Spring 2022
Introduction to Artificial Intelligence
HW Assignment #3: Due **Monday, April 18th 11:00am** (via the submission link on canvas)

Please see the accompanying video on the course Panopto site.

Building a Decision Tree

Rules for group work:

I suggest working in a GROUP on this assignment. You may work in groups of 2 students. You will submit only ONE assignment for the group. You may not "divide and conquer" this assignment. That is, all students must be present (virtually is ok) and contributing while any work on the project is being done. In other words, you must "pair-program" to complete the assignment. Along with the honor code, you must include, in a comment at the top of your assignment the following statement: "All group members were present and contributing during all work on this project." This statement will be treated in the same manner as the honor code, so please make sure it is present and true.

In this assignment, you will implement a binary decision tree classifier by implementing the DecisionTree class according to the specifications described below. Since the tree is binary it can learn data when every feature has exactly two possible values (true or false).

Provided Code

Your class should interact with the following classes that are already provided for you:

- TreeNode.java – represents a node in the decision tree.
- Example.java – represents a boolean feature vector for one example.
- TestClassifier.java – loads positive and negative training and testing examples, runs your decision tree class, and outputs the results.

You won't need to edit these files, but you should read and familiarize yourself with them so you understand how to use them. The code is located here:

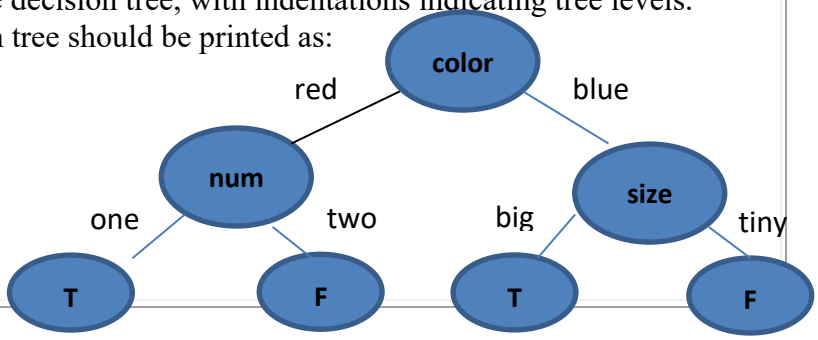
https://middlebury.instructure.com/courses/10232/files/folder/HWs/HW3/Part1

Part 1: Build the Decision Tree Classifier

You will write the DecisionTree class according to the following specifications:

DecisionTree – Fields	
`TreeNode`	**`root`** Represents the root of a binary decision tree.

DecisionTree – Methods

void	**train**(ArrayList<Example> examples) Builds a decision tree on the training examples in examples. This is the **Construct-Tree()** function we discussed in class.
void	**createChildren**(TreeNode n, int numFeatures) Creates the true and false child nodes for node n.
double	**getRemainingEntropy**(int f, TreeNode n) Computes and returns the amount of remaining entropy if feature f is chosen as the next node.
double	**getEnropy**(int numPos, int numNeg) Computes and returns the entropy of a node given its number of positive and negative examples.
boolean	**classify**(Example e) Uses the tree to classify the given example as positive (true) or negative (false).
void	**print**()//Already provided for you Prints a description of the decision tree, with indentations indicating tree levels. For example this decision tree should be printed as: color = red: num = one: T num = two: F color = blue: size = big: T size = tiny: F 

You are welcome to create any other fields or methods you find useful in your DecisionTree class and the given TreeNode class (do not modify the Example class). You are also welcome to change the return and parameter types of the methods. The only requirement is that **your classes must run with the given driver program TestClassifier.java**. Remember that it is always good programming practice to keep methods as concise and simple as possible by using the methods to do meaningful chunks of work.

If you have implemented the tree correctly, you are done with the programming components of the assignment (i.e. the challenging part), so congratulations! ☺ Read below to figure out how to know if you have implemented the tree correctly.

Part 2a): **Test Your Classifier**

To make sure your tree is working correctly, you will train and test it on four randomly generated datasets (set1-small, set1-big, set2-small, and set2-big).

Train and test your decision tree on the datasets, by running:

`TestClassifier` with `set1 small` as the program (command line) argument
`TestClassifier` with `set1 big` as the program (command line) argument
`TestClassifier` with `set2 small` as the program (command line) argument
`TestClassifier` with `set2 big` as the program (command line) argument

Each of these commands will train the tree on a set of positive and negative training examples, test the learned tree on a set of positive and negative testing examples, and report the results.

The datasets:
Each dataset is split into two separate files: one for positive examples the other for negative examples. Each training example has a value for each feature of the dataset. The features are numbered starting from 0 (so for example, feature 3 is the 4th feature). Here are the first few lines of the positive dataset for set1-small:

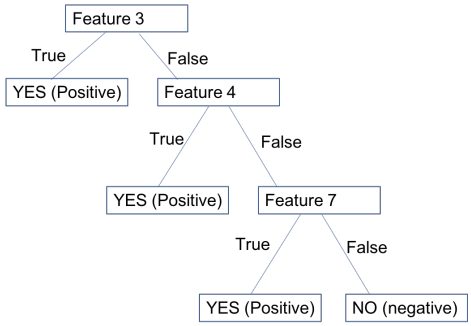
```
false false true false true true true true
false false false true true false true false
false true false true false false false true
…
```

The first line shows the values (either true or false) for each of the 8 features (feature 0 to feature 7) for example 1. The second line shows these values for example 2; the third line for example 3, and so on.

In these datasets all positive examples meet the following criterion:

feature 3 is true OR *feature 4 is true* OR *feature 7 is true*

So, the decision tree should be fairly simple – something like this:



However, because of our datasets, we may not end up with this decision tree. One reason is that some of the datasets contain *noise*, specifically some negative training examples also meet the criterion above. And even when the training data is noise-free, we may not end up with this decision tree (can you think of why not?). The goal is to use these datasets to investigate how well a decision tree can learn the criterion.

The only differences between the datasets are:

1. The small ones have 10 examples and 8 features while the big ones have 100 examples and 10 features.
2. Set1 is *noise-free*. This means that positive training examples always have the property specified above and negative training examples never do.
3. Set2 contains some *noise*. Specifically, positive training examples always have the property but about 10% of negative training examples also do.

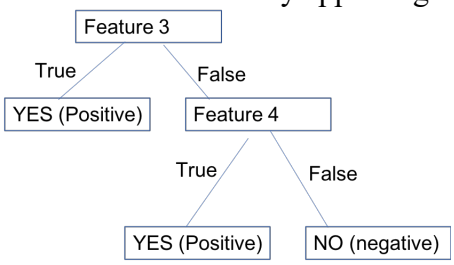
What your program should output:

The small sets are mainly for testing purposes. Here is the output (i.e. the decision tree) you should get:

- For set1-small:

```
Feature 3 = True:Positive
Feature 3 = False:
      Feature 4 = True:Positive
      Feature 4 = False:Negative
Positive examples correct: 9 out of 10
Negative examples correct: 10 out of 10
```

Here is a more visually appealing version of the tree:



- For set2-small:

```
Feature 6 = True:
      Feature 0 = True:
            Feature 5 = True:Negative
            Feature 5 = False:Positive
      Feature 0 = False:Positive
Feature 6 = False:
      Feature 7 = True:
            Feature 3 = True:Negative
            Feature 3 = False:Positive
      Feature 7 = False:
            Feature 3 = True:Positive
            Feature 3 = False:Negative
Positive examples correct: 4 out of 10
Negative examples correct: 5 out of 10
```

Your analysis will focus more on the big sets.
In your ReadMe, discuss how your decision tree performs on the big sets and briefly explain these results.

Pseudocode:

Below is the pseudocode for the `train()` method (i.e., the method for constructing the tree).

```
train(TreeNode node)

      1. If all remaining examples at this node have the same label L
         (Base Case 1):
      2.     -set this node's label to L
      3.     -set this node as a leaf
      4. If no more examples at this node (Base Case 2):
      5.     -set this node's label to the majority label of its
              parent's examples
      6.     -set this node as a leaf
      7. If no more features (Base Case 3)
      8.     -set this node's label to the majority label of this
              node's examples
      9.     -set this node as a leaf

      10.Else:
      11.    pos = node.getPos(); neg = node.getNeg(); // Get the
                        //positive and negative examples for this node
      12.    Find the next feature to split on, i.e. the feature, f,
             with the most information gain
      13.    Set this node's feature as f
      14.         -createSubChildren(node)//each node will have two
                   subchildren: a true child and a false child
      15.         train(this node's true child)
      16.         train(this node's false child)
```

Tip:

Debugging the training algorithm is somewhat tricky. The easiest way to avoid hard-to-find bugs is to build your program up incrementally, testing as you go. Try to break your training procedure into multiple procedures and test each of these procedures to make sure they work individually. For example, you might write functions that split a given dataset around a particular feature and test that it trains properly for this single feature.

Part 2b): Testing on Real-World Data

Now you will test the decision tree on a real-world dataset located here:

https://middlebury.instructure.com/courses/10232/files/folder/HWs/HW3/Part2

The dataset is derived from this one (http://archive.ics.uci.edu/ml/machine-learning-databases/hepatitis/hepatitis.names) and contains data about patients who were seen for ***hepatitis***. The goal is to predict whether or not a given patient survived. The dataset is already divided into a training set and a testing set. The ReadMe describes the features and provides the number of examples.

The class TestClassifierHepatitis.java is provided for you in the above directory. It loads positive and negative training and testing examples from the hepatitis data files, runs your decision tree class, and outputs the results.

(i) Run TestClassifierHepatitis.java on the hepatitis data to predict survival for hepatitis patients. **In your ReadMe**, describe how your decision tree classifies patients. Use Wikipedia or an on-line search engine to find out the meaning of the features whose names are unfamiliar to you. Do image searches at your own risk :-) Do not turn in feature definitions, but knowing a little about them may help you interpret the learned decision trees.

(ii) **In your ReadMe** report the percentage of correctly classified examples, the number of false positives and the number of false negatives.

Part 3: Testing your own dataset

There are many very interesting datasets publicly available to investigate. They range in size, quality, and the type of features and have resulted in many new machine learning techniques to be developed. For this part, you will find a public, free, supervised (i.e. it must have features and labels), machine learning dataset.
To find a dataset, you can try:

- googling "machine learning binary dataset"
- browsing the Kaggle repository: https://www.kaggle.com/datasetss
- browsing the UCI repository: http://archive.ics.uci.edu/ml/
- browsing KDnuggets: http://www.kdnuggets.com/datasets/index.html

You may have to modify the dataset to make the features binary valued so that they work with the decision tree you have implemented.

Once you've found the dataset, provide the following information **in your ReadMe**:
(a) The name of the dataset.
(b) From where you obtained the data.
(c) A brief (i.e. 1-2 sentences) description of the dataset including what the features are and what is being predicted.
(d) The number of examples in the dataset.

(e) The number of features for each example.
Partition your data into training and testing (at least 50 examples each), train your decision tree on the training data, and test it on the testing data. **In your ReadMe**, discuss how your decision tree performs on the data (as in Part 2(a), report the percentage of correctly classified examples, the number of false positives, and the number of false negatives).

Output:

We would like to grade your programs as efficiently as possible. So please make sure your program does not output anything other than what is asked for above. Points will be deducted for extraneous output/debugging statements.

ReadMe:

As always, please provide a ReadMe that includes the following:

1. The names of the students in your group.
2. Names of files required to run your program.
3. Any known bugs in your program (you will be less penalized for bugs that you identify than for bugs that we identify).
4. The discussion of results and data asked for in Part2 and Part3.

Submission:

Please include, in a comment at the top of every file you submit, the following:

- Full names of students in the group
- Group honor code: "All group members were present and contributing during all work on this project"
- Acknowledgement of any outside sources of help (discussions with other students, online resources, etc.)
- Middlebury honor code

Zip the following files and use the HW3 link on canvas to submit exactly **ONE** zipped file per group:

- TreeNode.java
- DecisionTree.java
- ReadMe

Name your zip file as follows: HW3_<last names of all group members>. So, for example, if your group consists of the last names Alvarez and Baker, you should name your file HW3_AlvarezBaker.zip.

Grading:

[77] - Correctness of implementation of above methods

[22] `train()`

[11] `classify()`
[22] `createSubChildren()`
[22] `getRemainingInfo()`

[7] - Comments in code
[16] – Discussion of Parts 2 & 3 in Readme

Written Problem

Submit on canvas one typed copy per group with the names of both students by the due date specified above. No emailed submissions.

[18] Use the algorithm we discussed in class (choosing the feature with the most information gain) to construct a decision tree for the following dataset. A_1, A_3, and A_3 are the features and the Output *y* is the label. Show the computations made to determine the attribute to split at each node.

Example	A_1	A_2	A_3	Output y
$\mathbf{x}_1$	1	0	0	0
$\mathbf{x}_2$	1	0	1	0
$\mathbf{x}_3$	0	1	0	0
$\mathbf{x}_4$	1	1	1	1
$\mathbf{x}_5$	1	1	0	1