**CS 662**

**Artificial Intelligence Programming**

**Project #1**

**Decision Trees**

**Due: Thursday, March 6, midnight**

**Summary**

In this assignment, you will be writing code to construct a decision tree, and then using it to classify instances from several different datasets. The decision tree algorithm is recursive; the amount of code needed is not huge, but there's some mental hurdles needed to completely understand what's going on – you’ll want to start this one early, as it’s more difficult than your previous assignments (that’s why it’s called a “project”). **This project should be done individually, not in your teams. Please re-read the section in the syllabus on academic honesty – remember, no looking at or sharing code. This includes code on the Internet.**  
  
I have provided some skeleton code to get you started and guide you through the implementation. The code is designed to help make the decision tree easy to implement and we need your method names & signatures to stay as they are so we can autograde (parts of) the code.  
  
**Datasets:**  
There are two types of datasets for this assignment (provided in the zip file)

* Toy datasets:
  + The tennis dataset
  + The restaurant dataset

These are both useful for testing your code; they're small, and you know what the correct answers are.

* "Real" datasets.
  + Lymph node data. This is the same lymphography.csv file that you worked with for HW1
  + Nursery school data. This data set contains nursery school applications for a large number of parents. Based on characteristics about the parents, we would like to predict whether the child should be admitted to nursery school.
  + IBM Watson dataset – getting the dataset working with all features for your decision tree is extra credit, as described below. I have provided a subset that contains only the binary features, and only contains ~23,000 examples instead of all 240,000 (tgmc\_stripReal\_subset.pkl – created using joblib and some preprocessing)

These datasets will be much more interesting for evaluating the performance of your decision tree. (More on that below.)

**Code**:

I've provided you with an updated version of <mlUtil.py>.

* It creates the enhanced dictionary with ‘id’ and ‘q\_id’ similar to contest.py that I posted earlier this week (but you don’t need to use this in the homework – it’s just for handy reference for the full contest data).
* It has a method extract\_data for reading in all the CSV files associated with the datasets above. NOTE: different datasets will need different arguments when calling extract\_data. If you can’t tell by looking at the csv file, please ask Cindi or Kaiming for help.
* For your decision trees, you will need easy access to all the values of each feature for categorical values. I have added a new method enhance\_data that adds a key to your data dictionaries that contains a dictionary of these values. Test it out on the tennis data to make sure you understand what it does. (Note: sometimes in the code I use the word attribute for feature. They mean the same thing). This dictionary will be needed by your decision tree constructor (\_\_init\_\_)

I’ve also provided you with a shell, <decisionTree.py> which contains the main class you will work with, DecisionTree. The main access methods for this class, as with ZeroR, are constructor, fit, and predict. Below are the steps you should follow for filling in this shell. Remember to test each method before moving on to the ones that build on the earlier ones!

A hint: list comprehensions are very helpful for this assignment. Often, you'll need to pull out one or more columns from the data. So, for example, to get a list containing only the third column in a dataset where the last element is equal to some item 'x', you could do:

third = [d[2] for d in data if d[-1] == 'x']

**Assignment:**

1. **(15%)** The decision tree code is easiest to code in a bottom-up fashion. To begin, we'll need a method to compute entropy. It should take as input a list of class values, such as ['yes', 'no', 'yes', 'yes'] and return a float indicating the information content (entropy) in this data. I've provided a function stub for you.
2. Next, we'll want to compute remainder. This will tell us, for a given feature, how much information will remain if we choose to split on this feature. I've written this one for you.
3. **(17%)** Once we know how to compute remainders, we need to be able to select a feature. To do this, we just compute the remainder for each attribute and choose the one with the smallest remainder. (this will maximize information gain.) The function selectAttribute should take as input a list of lists (the attribute-value vectors, **X**), and a list of target values, y. I've provided a stub for you.
4. We're now ready to think about building a tree. A tree is a recursive data structure, which consists of a parent node that has links to child nodes. I've provided a TreeNode class for you that does this. (You don't need a separate Tree class.)   
     
   The TreeNode has the following data members:
   * attribute: for non-leaf nodes, this will indicate which attribute this node tests. For leaves, it is empty.
   * value. For leaf nodes, this indicates the classification at this leaf. For non-leaf nodes, it is empty.
   * children. This is a dictionary that maps values of the attribute being tested at this node to the appropriate child, which is also a TreeNode.

It also has methods to print itself and to test whether it is a leaf.

1. **(4%)** So we need a method that can build a tree, or “fit” the tree to the training data. The first part (fit) just accesses any setup information from the constructor, and then call makeTree, described next.
2. **(22%)** We will call this makeTree (I have provided the method signature). It should work as follows:
   1. If the dataset contains zero entropy, we are done. Create a leaf node with value equal to the data's classification and return it.
   2. If the dataset is empty, we have no data for this attribute value. Create a leaf node with the value set to the default value and return it.
   3. If there are no features left to test, create a leaf node with value set to the majority class and return it.
   4. Otherwise, we have a non-leaf node. Use selectAttribute to find the attribute (feature) that maximizes gain. Then, remove that column for the dataset and the list of attributes and, for each value of that attribute, call makeTree with the appropriate subset of the data and add the TreeNode that is returned to the children, then return the TreeNode. You can ignore features whose value-list is ‘numeric’ rather than a list (see the extra credit question below)
3. **(3%)** Set up the constructor (\_\_init\_\_) to initialize any information you’ll need later on.
4. **(20%)** Now we know how to build a tree. We need to use it, though. To do this, you should fill in the fit() method in Decision Tree and the predict() method in TreeNode.   
   This method is also recursive. If we are at a leaf, return the value of that leaf. Otherwise, check which attribute this node tests and follow the appropriate child according to the example passed in. If there is no child for this value, return a default value.
5. Congratulations! You now have a working decision tree. Test it out on the toy datasets. You might find it helpful to build a better printTree method, although this is not required. You might also want to add code to pickle your tree to a file, and a main to allow you to easily specify options.
6. **(13%)** Once you are convinced your tree is working correctly, you should evaluate its performance. To do this, choose two of the "real" datasets. To evaluate the tree's performance, you will do 5-fold cross-validation. This means that you select 80% of the data to train on, and hold back 20% for testing. This should be done 5 times, each with a different randomly-selected training set. Fill in the stub for <eval_harness.py>
   1. For each validation fold, calculate the tree's precision, recall, and accuracy, and then average these across all 5 runs; you should do this for both the training set and validation (test set) accuracy. For the datasets with multiple classes (nursery, lymphography), you should choose in advance the most common target value as “positive” and the others should all be treated as negative, for purposes of P/R
   2. Run your test harness on ZeroR from the previous homework.
7. **(6%)** Prepare a short (3-4 paragraph) document that describes your tree's performance and discusses any anomalies or unexpected outcomes. You should talk about:
   1. Did one of the data sets prove more challenging than the others?
   2. What was the difference between training and test set accuracy? Was your tree badly overfitting on any of the data sets?
   3. How did your decision tree perform in comparison to ZeroR?
   4. Did it appear that any of the datasets were noisy or had other interesting issues?
   5. Did the trees look very different (or have big differences in the sizes) between the different folds?
8. **(5 points extra credit)** All the feature splits above are based on a discrete set of feature values and ignores the real-valued features. For extra credit (and the ability to handle the full IBM dataset), implement a version that also handles real-valued features. You might want to make a backup copy of your original decision tree code first to make sure you don’t break the required functionality above. Hand in a program called decisionTree\_numeric.py to have a chance at receiving this extra credit.

**What to turn in:**

* decisionTree.py: your version of all code described above through question #9.
* Tree\_eval.py: the code described in #10.
* discuss.txt (or .doc or .pdf): the document described in #11