Nick Jones
CS 361 - Intro to Evo. Comp.
Professor Sherri Goings
March 14, 2011

Code Documentation - GP Classifier

General Flow

While the purpose of this project was to explore genetic programming, a good deal of work went into creating the framework used to do so; about half of my code deals with parsing data from a datafile, organizing these data, and creating random decision trees.

gpclassifier.py is the main program in the entire algorithm. It initializes parameters, which are then passed to the run() method. The run method takes care of the rest of the algorithm, getting the data from the file that is passed in as a command line argument, creating an instance of the Algorithm class, and calling stepGeneration() for however many generations the algorithm should run. The overall structure of the program is to:

1. Gather and organize data from file (which is passed as a command-line argument)
2. Initialize Algorithm(), and its population, based on the data
3. Run a given number of generations

Step One:

Gathering and organizing the data is done almost exclusively within parsedata.py. The data should be in .csv format, with the first line of the file as labels for each category. Each line is read and stored as a dictionary of the form category => value; in other words, each line has a dictionary whose keys are the categories and whose values are the values on that line for the given category. parsedata.py returns this list of dictionaries, which are later used to form Entry() objects.

In addition, it returns the limits on each category in one of two forms. If category x is numerical, then the limits on category x are the minimum and maximum possible values; if category x is categorical, the limits are then every possible value in that category.

Finally, parsedata.py also returns the name of the category which we are trying to predict. In the original file, this category must always come last, and must be categorical (since this is a classification problem, not a regression problem). All these data passed to the main program are used in order to create randomly initialized decision trees.

Step Two:

Once all the data has been parsed from the file, an instance of the Algorithm class is initialized, which in turn initializes a population of Individual objects. Each individual in this implementation is simply a decision tree. In order to initialize an individual, a random decision tree also has to be initialized.

At first, each decision tree was created to be symmetric and full. However, when taking into consideration the number of possible successful trees which may be asymmetric and/or not full, it made much more sense to allow trees to be of varying structure and shape. Thus the initialization of a tree, call it X, proceeds as follows.

- Generate a random number between 1 and the maximum height of the tree - this number will be the maximum height of X.

- Pick a random category, and a random “cutoff” value for that category. The cutoff value is chosen randomly from the range of all possible values for that category (in the case that the category is a categorical variable), or chosen randomly in the range between the minimum and maximum possible values for that category (in the case that the category is a numerical variable).
- Create a new Node() object based on the random category and random cutoff value.
- This first node is the root of the decision tree.
- Place that node in a list of nodes that need to be “processed” (see below)
- Loop through the unprocessed node as follows:
    - In the case that the category selected was the class to predict, then the node just created is a terminal node, so it does not need (and cannot have) any children.
    - Otherwise, the node needs children - call the function to create two more random nodes, and make them the children of the current node (which is now considered to be “processed”); the function returns these children.
    - The new children are added to the list of unprocessed nodes (which no longer contains their parent)
    - Go back to the start of the loop, and repeat until all nodes have been processed (i.e. all the unprocessed nodes are terminal nodes) or the maximum tree depth is reached.
- Now the tree should be complete, and have the property that each internal node has two children (which may or may not be terminal nodes), and each terminal node has no children.

Step Three:

Now that an original population of random individuals has been created, the evolutionary process can begin. The best implementation seems to be based on a steady-state population; each generation, two individuals are selected randomly from the population to be parents. These two individuals are cloned, and then crossed over and/or mutated (each with an 80% chance of occurring). Finally, the two new children are compared to their parents; the fittest two individuals of the four (two parents, two children) survive and are put back into the population. This process repeats for some set number of generations.

Major Classes

Algorithm:

The Algorithm class is in charge of the actual evolutionary process. It contains everything that would be expected for a genetic algorithm - a population, crossover/mutation functions/rates, and so forth. In addition, however, the Algorithm class needs to know everything about the dataset. As a result, it also keeps track of all the Entry objects from the dataset, the categories, the class (category) to predict, limits of each category, and the maximum tree depth. All of these are necessary for the random initialization of GPDecisionTree objects, and therefore for Individual objects. In addition to having a method to initialize the population, the stepGeneration() method is the basis for evolution. In each call to this method, a new generation is passed. My implementation is based on a steady-state population. When a generation passes, two random individuals are selected to be parents, and cloned as children. These children then undergo crossover and/or mutation; the fittest two individuals among the parents and children are selected to survive in the population.

Entry:

The Entry class consists simply of a dictionary of the form category => value. Each entry corresponds to one line in the data file; each key in the dictionary is one of the data categories (i.e. height, weight, gender), and its value is that line's data value.

GPNode:

A GPNode can be thought of as an extension of any binary tree node. Each instance has a category and a comparison value (sometimes referred to as "cutoff" value). In addition, all internal nodes have a left and a right child, while terminal nodes have no children (their children are set as None). A node has the ability to be compared to some value in every entry: the value that corresponds to the node's category in the entry's dictionary. All numerical nodes are compared in the form entry's value < node's comparison value.

When a GPNode is actually evaluated, it takes in an Entry object. The node compares itself to the corresponding value in the Entry object's dictionary. If comparison returns false (i.e. entry's value > node's comparison value), then the evaluate function is called recursively on the node's left child. If comparison returns true (i.e. entry's value < node's comparison value), then the evaluate function is called recursively on the node's right child.

When a terminal node is reached, the terminal node's value (which is always categorical) is compared to the entry's value for the terminal category. If they are the same, then the tree has succeeded in classifying the entry correctly and True is returned; otherwise, the tree has incorrectly classified the entry, and False is returned.

GPDecisionTree:

The GPDecisionTree class is one of the most important, since each Individual has a GPDecisionTree object. Each instance contains a GPNode that is the root of the entire decision tree. The majority of the GPDecisionTree class is dedicated to initializing random trees; this process is discussed in depth in "Step Two" of the general flow section above. In addition, GPDecisionTree has an evaluate() method that is the basis for fitness evaluation. This method calls its root node's evaluate() method, which will recurse through the entire tree until reaching a terminal node. Upon attempting to classify all Entry objects, the method returns both the number correctly classified and the total number tested.

Individual:

An Individual object is very simple. All it contains is a GPDecisionTree, a fitness value, and a fitness function to reevaluate its fitness. The only fitness function used throughout is based on an individual's ability to correctly classify entries, and is further discussed below in the "Major Functions" section.

Other Major Functions

parsedata.py:

This function is a customized script for reading and interpreting the data from a .csv data file. Without going into too much detail (it's not the most interesting script ever written!), the function returns important information regarding the dataset:

- A list of dictionaries, each dictionary corresponding to one line in the file. Each dictionary is used to create Entry objects.
- A dictionary of category limits. Each key in the dictionary is a category (string); its corresponding value is the range of values within that category. If the category is numerical, then the range is

represented as [minimum, maximum] of the values found within the entire dataset for the given category. If it is categorical, then the range is a list of all possible values within that category (i.e. if the category were seasons, the range would be [fall, winter, spring, summer])

- The class/category the decision tree is attempting to predict - whether or not an applicant should be granted credit, whether a person has diabetes, etc.

getRandomNode():

I originally wrote this method for the purpose of aiding in the deletion mutation algorithm. However, it turned out to be very useful in basically all the different mutation and crossover functions, so it deserves attention here. The method takes in both a starting node and a maximum depth. It then loops until it either reaches a terminal node or reaches the maximum depth allotted. Each time through the loop, the current node is updated to go randomly left or right. This process continues, and once a terminal node is found or the maximum depth is reached, three things are returned.

1. The actual node to be replaced
2. The parent of the node to be replaced
3. Whether or not the node to be replaced was a left or a right child (True if left, Falsie if right)

Once these values are returned, whatever function that called getRandomNode() originally can replace the node by adjusting child pointers accordingly.

Fitness evaluation:

Fitness evaluation was simply based on how many entries a given individual's decision tree correctly identified. When the fitness function is called, GPDecisionTree's evaluate method is called for every Entry in the dataset; evaluate returns True if the entry is correctly classified, False if it is incorrectly classified. In most actual data mining applications, only a subset of the entire dataset is used as a "training set." I decided to use the entire dataset; since the purpose of this project was more concerned with the behavior of genetic programming and the datasets weren't overly large, there was no disadvantage to using the entire set.

Mutation/Crossover/Selection:

These are all described in detail in my report.

Output

Every 50 generations, a bunch of information is printed out regarding the population at the time. The best way to explain is with an example; here is one from a recent (very good!) run on the credit dataset:

```
Generation 736
Best ind's fitness: 0.872463768116
Best ind's height: 7
All heights in population: [7, 7, 7, 4, 7, 4, 7, 7, 7, 7, 7, 7, 7, 4, 6, 6, 7, 7, 7, 7, 7, 7, 5, 6, 7, 6, 6, 7, 3, 3]
```

It's mostly self-explanatory; the best individual's fitness is reported, in addition to the height of the decision tree of that individual. In addition, the heights of all the decision trees are printed to show whether or not bloating is having a large impact. Finally, every 200 generations, the best individual's decision tree is actually printed. It's a little incomprehensible at first, but the general form is [node [leftChild's subtree], [rightChild's subtree]].

The print statements that give these updates can be found in gpclassifer.py (the main program) in the run() method.
In addition, I tried to keep the parameters as separate from the rest of the code as possible. You should be able to set all of them at the bottom of gpclassifier.py, right above where I call the run() method.