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

Genetic Programming Classifier - Report

Goal:

My original goal was to implement genetic programming to solve an interesting classification problem. As my project progressed, however, I focused more on the reasoning behind the behavior of my classifier, and less on developing it to work well on large datasets. Thus I succeeded in implementing genetic programming, and also learned a good deal of the reasoning behind its advantages and disadvantages.

Original Hypotheses:

My two original hypotheses were somewhat vague, but I received valuable insight into both issues throughout the development of my classifier:

1. Explicit measures will be necessary to prevent bloating
2. I may need to exercise more control over processes (for example, initialization)

I didn't get a chance to explore fully the issues related to bloating or initialization. I ended up not using any explicit measures to control bloating; instead, I used an implicit measure that had relatively good results. Bloating and the implicit measure I used will be discussed in far more detail throughout the rest of my report.

Initialization turned out to be an interesting issue, particularly as it pertained to bloating. I found that the first "best" individual in the population had a large influence on whether or not bloating became an issue. In other words, if the fittest individuals at the start had "tall" trees, the next generations tended to bloat; with "short" fittest individuals at the start, bloating was less of an issue. Therefore, it seems that some sort of intelligent initialization (i.e. instead of random) would greatly benefit the overall drift of the population, and could prevent bloating. This is also discussed in more detail below.

Datasets, Some Results

Since they are mentioned later in the report, I will provide some data and background on the datasets I ued. Throughout the development of my implementation, I utilized three datasets. Here is a little information on them to keep in mind (they are all included in my zipped folder). I chose to use smaller datasets in order to focus more on the theory of what was going on as opposed to the practice of applying decision trees to larger sets. In addition, one of the papers I found regarding genetic programming as it is related to data mining used the same sets.

In Jeroen Eggermont's paper entitled "Data Mining Using Genetic Programming," Eggermont discusses increasingly more complex (and therefore more successful) implementations of genetic programming to classifiers. The values I compare are with his first GP, what is referred to as a *simple* GP in his paper.

Note that I am comparing my *best* results with Egermont's *average* results using his most simple implementation (there are quite a few better ones that he mentions later!). Also, credit goes to the UC Irvine Machine Learning Repository for providing these datasets.

Name of file	Number of Categories	Type of Data	Number of Entries	Average Classification Rate (Eggermont)	Best Classification Rate (my implementation)
iris.csv	4	Numerical	150	94.4%	98%
credit.csv	14	Num. and Cat.	690	78%	87.5%
pimadiabetes.csv	8	Numerical	768	73.7%	76.6%

Description of EA:

Representation:

Each individual in the population is simply a decision tree. This tree is not necessarily symmetric or full (i.e. subtrees are of varying heights). However, every node within the tree is either an "internal" node or a "terminating" node. If it is internal, then the node must have two children, which can be either other internal nodes or terminating nodes. If the node is a terminating node, it has no children by definition.

Fitness:

The fitness of an individual was simply the percentage of entries correctly classified. Since the datasets I used were not obscenely large, I trained each decision tree on the entire dataset. Thus, each entry was classified based on the individual's decision tree; the fitness was just the number of entries correctly classified divided by the total number of entries.

Initialization:

Since bloating is a large problem with genetic programming, it doesn't make sense to initialize individuals to their maximum heights. In addition, when my algorithm is applied to a simple data set, the best solutions are often only a few nodes "tall," and a decision tree of height 7 or 8 is unnecessary. In order to keep the initial solutions diverse, each individual in the population is initialized with one of four heights, which are spread out between 0 and the maximum possible height (i.e. if the maximum is 8, then each tree could be either of height 2, 4, 6, or 8). This could also be done randomly.

The actual tree is initialized in a random manner, as well. A node's attribute is selected at random from the list of all possible attributes; a "cut-off" value is then selected randomly from within the range (between the minimum and maximum) of values for the given attribute, if the attribute is numerical. If it is categorical, then the "cut-off" value is randomly selected from all possible values for the given attribute. See the Code Documentation for more detail on initialization.

Mutation:

This was one of the major challenges of the project. Since there are two aspects of a decision tree that need to be altered, mutation has to address both of these. The first is the actual structure of the tree - how tall should it be? What should it shape be like? How many internal nodes does it need? What are the best predictor attributes, and how early in the tree should they be included? The second aspect is the "cutoff" values within each node. When the decision tree is processed, each entry in the original dataset is compared to these cutoff values - what value will tell us the most? For example, does a person's height

being greater than 5 feet tell us something about their chances of playing basketball? What about 5 foot, 5 inches? Figuring out appropriate cutoff values is a problem that must be addressed throughout the program, and especially in mutation.

Since there are a few different aspects of the tree that need altering, I wrote multiple mutation functions -

- Deletion of a random node (and its subtree); it's replaced by a new, random terminal node
- Altering of a random node's cutoff values (by a Gaussian that is 20% of that node's attribute's range)
- Replacement of a random node with a new random subtree

The first serves multiple functions, but is most important in implicitly dealing with the issue of bloating. The second attempts to deal with the problem of finding an appropriate cutoff value for each node. The third serves to introduce new, random variation into the tree.

In order to incorporate all three (because they serve very different roles), each has a probability of being called each time an individual is mutated. The deletion method is used automatically if the individual being mutated is of too great a depth (is "too tall"). Otherwise, there is a 25% chance of the individual being mutated by the second mutation operation (cutoff value alteration), a 25% chance of the deletion operation, and a 50% chance that a random node is replaced with a new, random subtree. While the other two serve important roles, the subtree replacement is essential to maintaining diversity - it introduces completely new nodes into the individual, and therefore into the entire population.

Since mutation is essential to maintaining diversity in the population, the mutation rate is very high - 80%. In addition, my current implementation is based on a steady-state population. Given that parents are compared directly with their children, it's somewhat pointless to create children that don't mutate or crossover - if the children don't mutate or crossover, then they're the exact same individuals as their parents, and there's no chance for improvement. As a result, I have kept the mutation rate (and crossover rate) very high.

Crossover:

Crossover is generally thought of as the most important operation in genetic programming (some GPs don't even use mutation). My implementation stresses mutation, but also includes a lot of crossover (80% as well) in hopes of combining "useful" nodes into a better decision tree. The crossover function is pretty simple - a random node is selected from each of the two parents; the two nodes are swapped, along with their subtrees. While mutation attempts to create more diversity in the population as a whole, crossover is meant to combine the best nodes within individuals to create an optimal decision tree.

Selection:

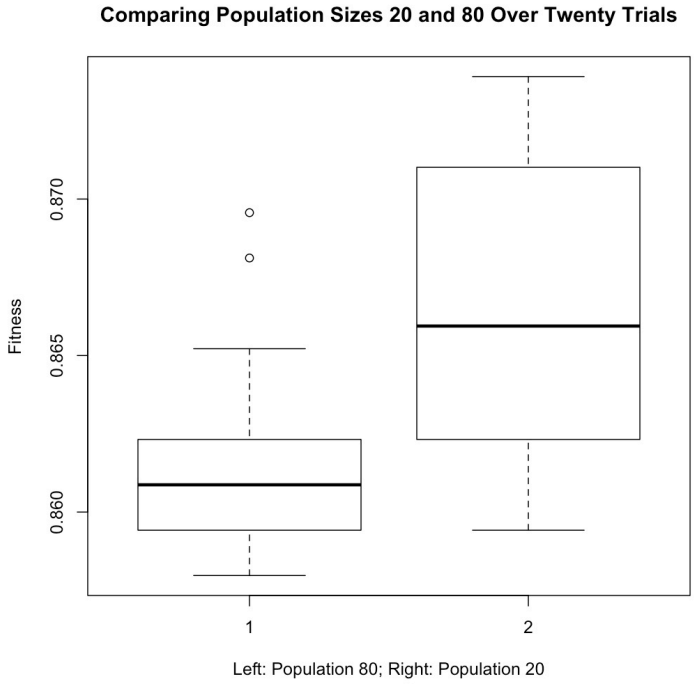
Since I'm using a steady state model, the selection process is deterministic. Each time two children are created (and mutated and/or crossed over), both children and both parents are removed from the population. Once mutation and crossover have (or haven't) occurred, the fittest two individuals of the four (two parents and their two children) are added back into the population. Therefore there is no chance for less fit individuals to survive - hence the process is deterministic.

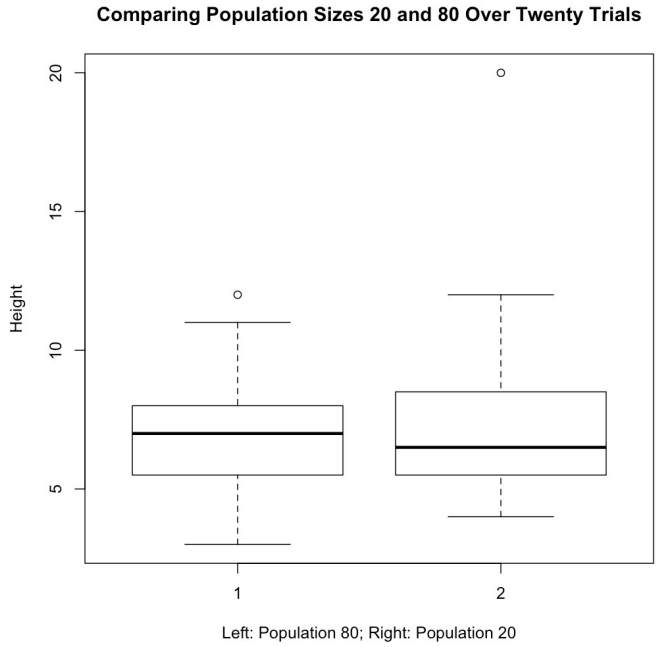
Population Size:

Determining the optimal population size for my implementation was difficult to do. As is true with any genetic algorithm, there are numerous parameters that have to be set, some arbitrarily due to a lack of data. In the case of population size, I considered two main factors: speed and diversity. It seems at first that each generation takes longer to run with an increase in population size; however, with a steady-state population, only two parents are selected each generation. As a result, it seems as though a larger population would be preferable. In addition, a larger population (at least hypothetically) seems less likely to succumb to bloating, since there are a larger number of individuals (given the initialization as described above) that are "short" to begin with. However, my short testing proved otherwise.

I tested these hypotheses on the credit dataset (see below for graphics), and determined from these results that a smaller population seems to work better. With that said, my testing was far from extensive - other factors (i.e. the number of entries, the number of categories, whether the categories are numerical/categorical) could play a role in the success of any given run.

I compared (over just 20 trials) the results of using a population size of 20 and a population size of 80 on the credit dataset. I let both run for 750 generations, and compared the best fitness levels reached, and the height of the best trees. Surprisingly, I found very little difference in tree height (second graphic), and a more significant difference in performance (first graphic):





Bloating

In genetic programming, one of the biggest and best known issues is bloating - as many generations pass, the decision trees tend to get really big. Some implementations attempt explicit measures to prevent bloating (i.e. automatically deleting subtrees of a tree that is too large), while others use more implicit measures. My implementation tries to deal with this issue implicitly with the mutation function I described above.

Though this generally works just fine, there are times that the trees do indeed bloat in size. Having watched many generations of trees pass, I have found that initialization plays a large role in whether or not individuals bloat. If my algorithm happens to find a pretty good solution that has a "short" tree, the population will remain within a reasonable range (i.e. won't bloat). However, if the initial population contains a relatively "tall" decision tree that is the best solution at the time, the population will drift towards larger, more bloated trees. This makes sense in the context of genetic algorithms in general; the population tends to converge to the fittest individuals. However, it's possible that exhibiting more control over the initialization process (i.e. using a different algorithm to generate pretty good, "short" trees) would have a drastic effect on the bloating problem.

Reflections on GP, this Project etc.

I think my implementation of genetic programming as applied to creating decision trees was relatively successful. One of my original goals was to get my classifer to run as well as the "simple" GP one discussed in J. Egermont's "Data Mining Using Genetic Programming" (which can be found here - http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/eggermont_thesis.html). As shown at the beginning of the report, my implementation succeeded in doing so.

Though we didn't have a chance to work with genetic programming in any assignments, I ran into a lot of the same challenges that are present in any genetic algorithm. I think that this project helped broaden my perspective on the number of applications the field of evolutionary computing can have. While most of the problems we discussed in class were optimization problems, a classifier is a completely different kind of optimization yet evolutionary algorithms can still be used.

The typical challenges of genetic algorithms definitely applied to this project, as well. The fact that an individual's representation was an entire decision tree added a multitude of parameters that needed to be set. How large is too large a tree? How should trees be initialized? Should I impose explicit measures to prevent bloating? In addition, the usual parameters - population size, mutation rate, crossover rate, and so forth - were challenging to determine. The representation also added an enormous amount of complexity to the search space. Whereas in some genetic algorithms we can at least consider the search space as being some combination of numbers, the number of possible decision trees is almost unfathomable. How many nodes is optimal? What order should there be amongst them? What about cutoff values? Is there any one optimal solution? These questions added further complexity to the problem of writing mutation and crossover operations. Is it possible to write such operations that are both connected and reachable?

Though my classifier was relatively successful, the field of machine learning and data mining generally takes a much more deterministic and algorithmic approach to classification. I mentioned above that initialization played a large role in determining what "optimal" solution was found. What if we used data mining techniques to create "pretty good" initial solutions, and then evolved these trees? It seems that this approach could be very successful, in that it incorporates both the algorithmic aspect of most machine learning techniques, along with the stochastic nature of genetic algorithms.