Symbolic Regression with Genetic Programming

Andrew Miller and **Michael Robertson**
{amiller,mirobertson}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

In this paper, we use genetic programming with expression trees in order to model an unknown dataset. The implementation liberates the model from regression selection bias and instead, relies on the a simulation of evolution through natural selection in order to converge on a best-fit function. For the first set of $(x, f(x))$ pairs, our program converged upon a highly accurate model. Though we didn't reach the same level of convergence for the second dataset of input output pairs of a multivariable function $f(x1, x2, x3)$, we seceded in moving toward a function with lower error, and we offer suggestions for future steps.

1 Introduction

In this experiment, we were given three datasets of 25,000 points and our goal was to use genetic programming with symbolic expression trees in order to generate an expression that best models the data and can accurately predict new data points. The expression tree format allowed us to preserve or alter parts of expressions through crossover and mutation across generations in order to develop better-fitting expressions. Dynamic Expression Trees have been studied by many, including Ferreira and Gepsoft. Expression trees are made up of operator, constant and variable nodes that form expressions when printed by traversing in order. These nodes act as genes in our genetic algorithm as seen in Figure 1.

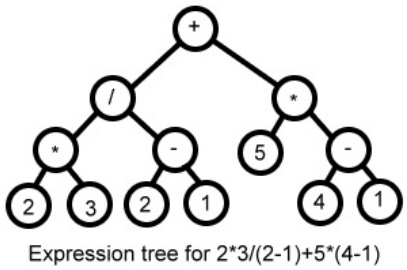

Figure 1: An example expression tree courtesy of `Stack Overflow`.

When implementing a genetic algorithm, in order to develop a symbolic regression solution to our datasets, we had to make decisions concerning selection methods, bloat control and fitness evaluation. All of these factors have been previously studied and we based our work off of the work of Poli, Langdon and Blickle and Thiele.

2 Background

Preprocessing

Before running our genetic algorithm, we chose to split our datasets into training and testing data using the 80-20 rule of thumb. We trained our trees on the test set, and then found the mean square-root error of the best-performing tree in the training set on the test set. Because we wanted our equation to model the underlying relationship between the variables, rather than simply interpolate the data point, we evaluated the quality of our solution based on its performance on data it had never seen before. Without this step, we could have simply over fitted the data, by modeling trends or relationships present only in the dataset we trained on, which did not actually exist between the variables.

Fitness

We then ran our algorithm using a starting population size of 500 randomly generated trees. The fitness of trees was evaluated using a combination of root mean-square error and a penalty function proportional to the tree's size in order to deter the tree from containing redundant or purposeless equations. The *bloat* problem of generating trees with large, non-functioning subtrees appears often in symbolic regression, and we used a method inspired by Poli in order to control it (Poli 2003).

3 Experiments

The difficulties of genetic programming present themselves to the designers in the form of infamously many adjustable parameters, all of which influence the end efficacy of the program in interdependent and inscrutable ways. We attempted to find a choice of population size, method of original tree generation, tree size penalty, mutation rate, mutation method, parent selection method (which itself has a own considerable set of variable parameters), and reproduction mechanism which both cause the program to converge to a solution and to maximize the rate of that convergence.

Definitions

We defined convergence as reaching a below 0.1 mean-squared error from the test dataset, and failure to converge as moving through 50 generations without producing a new child more fit than any of its ancestors. We measured the success of our program by the root mean-squared error of our best tree on the test data, which it had not previously seen.

To measure fitness, we used an two effective fitness function f_1 and f_2 for datasets one and two, respectively. We defined them as

$$f_1(x) = s \cdot size(\text{tree}) + mse(x)$$

(where s is a weight penalty, and the size of the tree is defined as the number of nodes in it) and

$$f_2(x) = k \cdot size(\text{tree}) \cdot rmse(x) + rmse(x)$$

(where k is a percentage, and $rsme(x)$ is the Root mean squared error of the tree)

Methods

Crossover Many methods exist for choosing crossover pairs. We chose to use tournament selection to determine the breeding pairs of trees. For dataset one, we implemented a basic tournament selection method, and we adapt work from Langdon n dataset 2.

When choosing the size of the each tournament, we face a trade-off between diversity and selection pressure. For large participation in each tournament, only one tree wins, so there is a high selection pressure, but diversity quickly falls. For participation in each tournament, diversity remains high, but more trees survive to reproduce, so selection pressure falls. For example, in a population size of 100, a tournament participation size of 50 (2 tournaments with 50 participants) allows for two highly fit victors, and thus quickly shrinks the genetic diversity of the following generations. Conversely, a tournament participation size of 5 (20 tournaments, with 5 participants each) allows many individuals to survive, but only requires them to be more fit than four others to do so. In general, we see that

$$t \cdot p(t) = s(g)$$

where t is the number of tournaments, $p(t)$ is the number of participants in each tournament, and $s(g)$ is the size of the current generation. Second, we ran fitness proportion selection, where we chose different methods for assigning reproduction probability.

Tree Generation We generated binary trees of size 6 that were built breadth-first. Thus, all of our trees in the initial populations were balanced. However, the crossover methods frequently mated trees of different sizes in dataset 1, and in dataset 2 the modified reproduction methods often changed the structure of the tree. Therefore, within the first two generations our population diverged from a balanced tree-set to a diverse set of tree shapes.

4 Results

Dataset 1

For dataset 1, with a initial tree size between 3 and 6, initial population size of 500 and a mutation rate of 15%, we achieved convergence on the tree seen in Figure 2 using a basic tournament selection method. The resulting expression tree, which performed with a root mean squared error of 0.024, modeled the data as a quadtradic function.

$$f(x) = x^2 - 6x + 14$$

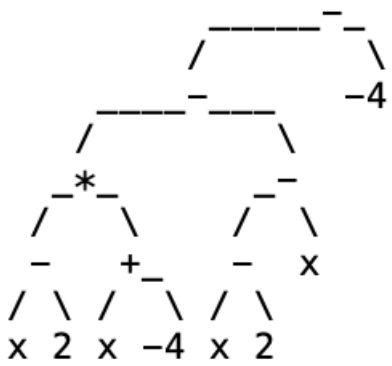

Figure 2: The resulting tree from achieving convergence on dataset 1.

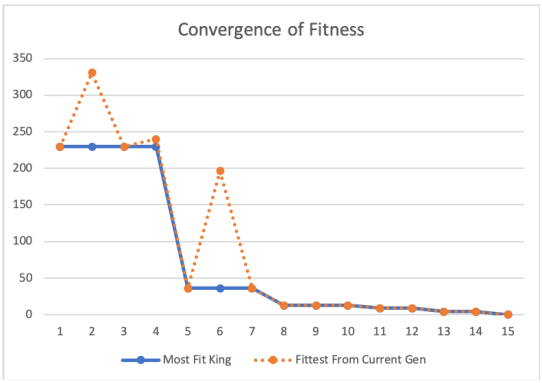

Figure 3: The path of convergence on dataset 1. The solid line represents the fitness of our fittest king overall, while the dashed line represents the fittest king at each generation

In Figure 3, we see that convergence was slow through the first 4 generations before making a leap to a much better fit in generation 5. This can be attributed to the random initial states of individual trees and the reliance on both crossover and mutation to find a closer approximation. We see that once our fitness made the significant improvement, after generation 6, the most fit tree from each generation remained the same or improved from the generation prior. Our algorithm converged in the 15th generation with a root mean-squared Error of 0.00771445. The remaining error can be attributed to rounding error, since the dataset we were working with was only precise to two decimal places.

Dataset 2

Dataset 2 presented a new challenge of a multivariable function. Our runs were converging slowly and yielding unfavorable results. We decided to examine the crossover method and bloat control that we had implemented for dataset 1. We ran our algorithm with initial population size of 500 and mutation rate of 5% using our original tournament selection process. We see in Figure 4 the use of the size-penalty function was causing a very slow convergence rate. Whereas, removing the size penalty yielded closer approximations within the first 20 generations. We expect that the size penatly reduced the diversity of our population, and didn't allow the random elements of our program to wander enough before a the entire population became dominated by the offspring of a sub-par model of the data.

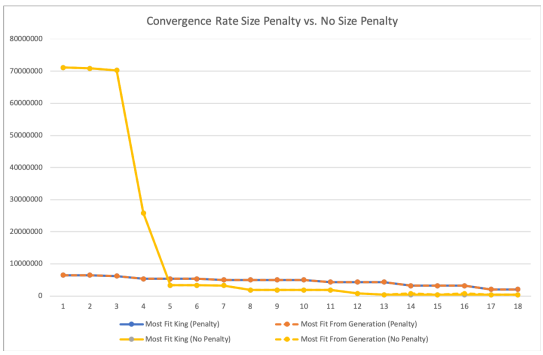

Figure 4: The difference in convergence with and with out including a size penalty in our fitness calculation. The test without size penalty (Yellow Line) comes draws nearer to a better approximation in early generations than the test with a size penalty included (Orange Line).

Additionally, we tried crossover methods from Kinnear and Jr.. We implemented five different types of reproduction, which occured with probability .35 or .10

1. **Fitness selection** (35%) chooses parents from the winners of the tournaments
2. **Non-fitness** (35%) chooses parents from the original population, without regard to tournament performance
3. **Standard Mutation** (10%) mutates a tournament winner by adding a randomly generated tree to one of its operation nodes
4. **Hoist** (10%) takes a random operator node within a tournament winner and places the subtree of that operator as a new individual in the next generation (thus "hoisting" a child node up to the root
5. **Create** (10%) places a randomly generated tree into the next generation

We expected these modifications in our reproduction method to increase our diversity and hasten our convergence, but when we trained the program on 80% dataset 2, our root mean square error began in the range of 10E18, and did not improve over 100 generations. Additionally, the diversity of our population, measured by the variance (σ^2) of the fitness values,

decreased from an order of $\approx 10^{20}$ to 10^{30} in generations 1 to 50, until it reached 10^{11} in generation 53, where it remained constant, along with the fitness, past generation 100, indicating that our population had lost its genetic diversity

To improve diversity, we increased the mutation rate to 20%, and removed Kinnear and Jr.'s modifications to parent selection.

The closest expression we were able to achieve for dataset 2 was:

$$\frac{x_1}{x_3^2 + \frac{1}{3}(x_3^3 \cdot x_2)}$$

This function had a root mean-squared error of 1229205.5577207913 on our test data set. As we repeated trials, we noticed that our algorithm was tending towards trees that only utilized one of the inputs, namely x_3. This homogeneity issue contributed to our difficulties with convergence.

5 Conclusions

We achieved convergence on an expression that yielded a root mean-squared error of 0.00771445 when compared to the test data for dataset 1. We attribute this error to lack of precision in the given dataset and conclude a successful fit.

For dataset 2, we were unable to converge on a best-fit function. We faced challenges with homogeneity and experienced bias towards a single input variable. Going forward, we would continue to tune the fitness function, adjust the mutation process to improve diversity and continue experimenting with alternative crossover and selection methods.

6 Contributions

Due to the large volume of coding in this project, the authors worked separately to develop moths of the classes and methods. Andrew wrote most of the functionality of the Tree, including the crossover, mutate, evaluation, and calculation of fitness methods. He also wrote functions to read the supplied data into a usable format. Michael completed the basic implementation of the Tree that Andrew began, as well as the underlying Node class. Michael wrote the driver program to create new generations, move through generations, as well as the two implemented tournament selection methods. Each partner researched the parts of the program they implemented. Andrew wrote the Abstract and Introduction. Michael and Andrew collaborated on the Background, Experiments, Results and Conclusions.

References

Blickle, T., and Thiele, L. 1994. Genetic programming and redundancy. *choice* 1000:2.

Ferreira, C., and Gepsoft, U. 2008. What is gene expression programming.

Kinnear, K. E., and Jr. 1993. Generality and difficulty in genetic programming: Evolving a sort.

Langdon, W. B. 2000. Size fair and homologous tree crossovers for tree genetic programming. *Genetic programming and evolvable machines* 1(1-2):95–119.

Poli, R. 2003. A simple but theoretically-motivated method to control bloat in genetic programming. In *European Conference on Genetic Programming*, 204–217. Springer.