Symbolic Regression Using Genetic Programming

Ilina Navani and **Katie Kowalyshyn** and **Oma Hameed**
{ilnavani,kakowalyshyn,omhameed}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

In this paper, we implemented symbolic regression using genetic programming on two data sets. Dataset 1 represented an equation with one variable, while Dataset 2 represented an equation with three variables. We found that for the first data set, our predicted function was $x^2 - 6x + 14$ with a mean squared error of 0.0006. For the second data set, our predicted function was $-1089/(6.5321x_3)^2$ with a mean squared error of 1.05 x 10^9. While Dataset 1 converged with logic based parameters, Dataset 2 required testing more specific parameters and yielded far worse results. This paper investigates implementation techniques as well as parameter settings used the symbolic regression problem, in order to yield the best possible results.

1 Introduction

Our goal was to solve the problem of symbolic regression using genetic programming on two data sets. Since the functions representing either data set were unknown to us, we were tasked with using symbolic regression to reconstruct them. Symbolic regression is a search algorithm that finds a mathematical expression in order to determine an equation that best fits the data provided. To implement symbolic regression, we used structures previously developed in computer science, specifically genetic programming and expression trees. Genetic programming is a technique of artificial intelligence that relies on evolutionary algorithms and natural selection to find a best-fit solution for a given problem. In order to use this method, we began by generating a random population of expression trees and then running a deterministic algorithm repeatedly to find the tree of best fit for the given data sets.

Structurally, each individual in our population was a binary tree representing an equation. Expression trees use leaf nodes to define operands that are either a variable or a constant, and use parent nodes to define operators such as multiplication, division, addition, and subtraction. Figure 1 depicts one such example of an expression tree that represents an equation evaluating to the value 25. Our expression trees, however, utilized variables as well and could therefore evaluate to expressions such as $x^2 + 4x + 3$.

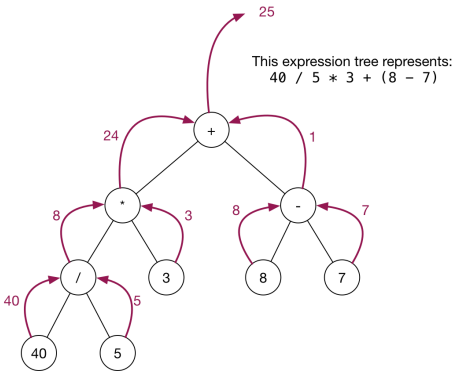

Figure 1: Expression Tree Example
Colorado State University

Our implementation of genetic programming runs for multiple generations within multiple evolutionary runs, selecting individuals from a constantly evolving population, until we are able to find a single best-performing individual. When developing these algorithms, we researched potential selection methods and data diversity control measures via various research papers. Two papers were especially of use. Specifically, we chose our selection mechanisms, a tournament-style decision process and a roulette wheel selection, from an the article written in 2018 by Ryan Champlin about various selection methods used in genetic programming for symbolic regression (Champlin 2018). Additionally, we learned a lot about combating issues of bloating and overfitting in our data sets from a paper called "Dynamic Maximum Tree Depth" by Sara Silva and Jonas Almeida (Silva and Almeida 2003), as well as from "Preference-driven Pareto front exploitation for bloat control in genetic programming" by Jiayu Liang, Yuxin Liu, and Yu Xue (Liang 2020). In the remainder of this paper, we will provide background, describe our experimental setup, and analyze our results.

2 Background

Data sets

The two data sets used in this experiment had 25,000 data points each. Dataset 1 had a single variable x with the corresponding values for a function $f(x)$. Dataset 2 had three

variables x_1, x_2 and x_3, with the corresponding values for a function $f(x_1, x_2, x_3)$. The fitting task for each data set was to determine its mystery function using symbolic regression, with the only difference being that $f(x)$ can be composed of operators $+, -, \times, \div$ and integer constants, while $f(x_1, x_2, x_3)$ utilizes real-valued constants in addition to the same set of operators.

Fitness Function

Every tree in our population had a fitness associated with it. Fitness is defined as the mean squared error between our model's predictions and the true function's values in a float value form. We calculate this by using the following equation, where we average the squared difference between the estimated values and the actual values across all data points. The equation is as follows:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Figure 2: Mean Squared Error (MSE) Equation

In symbolic regression, lower fitness values are better, because we want to get as close to the true value as possible. Therefore, in theory, the best-fit expression tree should have a fitness value of exactly zero.

Selection Mechanisms

To select each tree that underwent cloning, mutation or crossover in each generation, we used tournament selection as our selection method. In our implementation, tournament selection works by picking a random set of individuals from the total population, and determining which individual within that set has the best fitness. The size of the set is a parameter than can be set by the programmer. This selection method helps to preserve diversity by avoiding simply picking the best individual from the entire population each time (Champlin 2018). Additionally, for Dataset 2, we included a second selection mechanism called roulette wheel selection. Essentially, roulette works by converting fitness values into selection scores which are used to create a probability distribution where individuals with lower scores have higher selection probabilities. A random number is then generated, and the individual whose cumulative probability exceeds this number is chosen, ensuring a higher chance of selecting individuals with lower fitness. This approach further helps improve and preserve traits associated with better fitnesses (Champlin 2018).

Combating Bloating

To control for bloating, we decided to implement a penalty for trees that faced bloating. We found that the most common approach is to place limits on the depth or size of the evolved solutions, however, it is difficult to set a reasonable limit: if the limit is too low, we may not be able to find a satisfactory solution. Inversely, if it is too high, evolution will slow down due to high computational costs, and the chance of finding small solutions will be lowered. Having considered this issue, researchers have generated GP-based methods that reject children with a depth more than 17 (Liang 2020). Therefore, our fitness function assigned a fitness of infinity to trees of depth 17 or greater, essentially disregarding them from within the population.

Combating Overfitting

To mitigate overfitting, that is, to ensure generalization of our model, we partitioned our data sets using train-test splits of size 80%-20%. When running genetic programming, we were able to train our individuals on the training set and pick a best overall performer, and evaluate its performance on an unknown test set. Such a set-up helps mitigate overfitting by verifying that the best-performing tree represents the underlying relationship between the variables, and is not simply memorizing the data provided to it.

3 Experiments

Our experiments sought to find the best fitting equation for the given data sets by using genetic programming with an optimal combination of parameters.

Tree Generation

To structure our program, we represented each equation as a binary expression tree as shown in Figure 1. Each operation in the expression was a node in the tree, with leaf nodes being operands and all other nodes being operators. To initialize our initial "seed" population of trees, we generated random, full binary trees of a specified depth from 0 to 3, so as to restrict making them overly complex at the beginning. To create diversity in the initial population and encourage non-zero depths, we assigned higher weights to depths 1 and 2, thus resulting in more initial trees at these depths. For trees in Dataset 1, we randomly chose between operands which were integer values between -10 to 10 or the variable x, based on the nature of the data. For all non-leaf nodes, we randomly chose between operators $+, -, *$, or $/$. For Dataset 2, we randomly chose operand leaf nodes from a range of real numbers between -10 to +10 and between the variables x_1, x_2, and x_3, and for all other nodes, we chose between the same set of operators used in Dataset 1. After growing our initial population of 500 trees, we were able to run genetic programming for multiple generations to retrieve results with the best convergence and fitness.

Dataset 1 Parameters

Our percentages for mutation rate, crossover rate, and cloning rate were 20%, 65%, and 15% respectively, as these were the percentages for most optimal results. Crossover maintains the highest level of diversity in the population by choosing two trees, crossing them over at randomly specified nodes, and generating a new, crossed over child. Therefore, it has the highest probability of being chosen. Mutate has the second highest probability because it makes small incremental changes to leaf nodes, specifically, changing an x to a constant or leaving it as is, and changing a constant to an x or to an another constant different than its current value. Lastly, cloning simply copied over a tree to the next generation. Since each process generated one new individual, we maintained a population size of 500 for each

generation.

Our tournament size was 20% of the population. Specifically, for a population size of 500, a random pool of 100 individuals were chosen, and the fittest amongst them was selected for either crossover, mutation, or cloning. Upon trying larger and smaller values, we found that a tournament size of 20% gave us most optimal results.

To ensure reproducibility of results obtained via genetic programming, we set the number of independent evolutionary runs to 10. Within each evolutionary run, we originally ran our data for 100 generations, only to discover that our trees were converging to a consistent fitness much sooner than expected. Hence, we reduced our number of generations to 20 to stop genetic programming soon after our population converged. The tree obtained at the end of the 20 generations was selected as the best-performing tree in each evolutionary run. To determine an overall best-fit tree, we picked the tree that had the lowest fitness among all 10 independent runs. Finally, we ran the chosen best-fit tree on the test set and recorded its fitness.

The table below provides an overall summary of the final choice of parameters we used to determine results.

Parameter	Value
Crossover Rate	65%
Mutation Rate	20%
Cloning Rate	15%
Train-Test Split	80% - 20%
Maximum Initial Population Depth	3
Bloating Tree Depth Penalty	17
Population Size	500
Number of Generations of Evolution	20
Number of Independent Evolutionary Runs	10
Tournament Size	20%

Table 1: Summary of Parameter Values

Dataset 2 Parameters

For the second data set, we began by running genetic programming using the same set of parameters as Dataset 1. Since Dataset 2 had three variables, x_1, x_2, and x_3, our initial program had to be slightly modified to account for evaluation for all three variables. However, given the complexity of the data, we also anticipated a lot more overfitting and bloating issues. In order to proactively combat this, we implemented additional methods in an attempt to further improve diversity of our population and better fit the data. The first method was a new selection mechanism, roulette wheel selection. Within each generation, when selecting the best-fit individuals to crossover, mutate or clone, our algorithm had a 50% chance of using tournament selection and a 50% chance of using roulette wheel selection. Such an implementation encourages diversity, because the two different selection mechanisms pick the best trees in different ways.

A second method we attempted was to alter the process of mutation of nodes. Instead of simply changing the value of a leaf node, in certain cases, we not only changed the value but also inserted a newly generated subtree. Specifically, if a leaf node had a value of x_1, x_2 or x_3, we replaced it with a randomly chosen operator, and added a left child containing the original x-value, and a right child containing a real number from the specified range. Doing so allowed us to create more significant mutations, and therefore more diversity, within individual trees of the population.

4 Results

After running genetic programming on our two data sets with various parameters, we reached results for both. In Dataset 1, our results converged to a best-fit equation with a very low fitness within nine generations, while in Dataset 2, our fitness values remained much higher across generations, and never accurately converged.

Dataset 1

For this data set, our best tree converged at $x^2 - 6x + 14$ with an MSE of 0.0006 after nine generations of an evolutionary run. The chosen expression tree, which represents the equation, is shown in Figure 3 below, followed by our equation plotted along the actual data points in Figure 4.

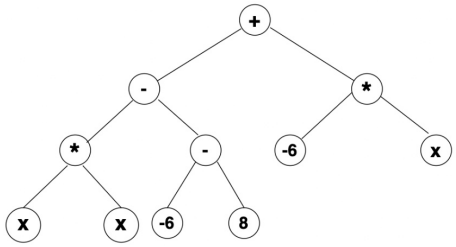

Figure 3: Dataset 1 Best-fit Expression Tree

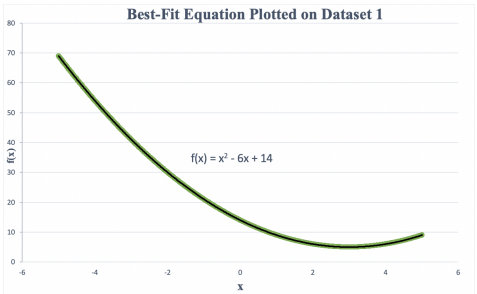

Figure 4: Best-fit Tree on Dataset 1
(data points - green; equation - black)

In the above figure, we see that our model's chosen equation represents the data very well. Given that the true $f(x)$ values in the data set were rounded to two decimal places, we were never able to reach an MSE of exactly zero. Nonetheless, our results for this data set were fairly consistent, converging within ten generations for most runs using the same set of parameters. Figures 5 and 6 show a generation-by-generation progression of fitnesses in our best evolutionary run. As seen in these figures, fitness values dropped significantly after two generations, and this was a trend we observed over most of our runs. Finally, on the test set, our best-performing tree had a fitness of 0.0006, indicating that it did not overfit the training data and does indeed represent the relationship between variables in the data set.

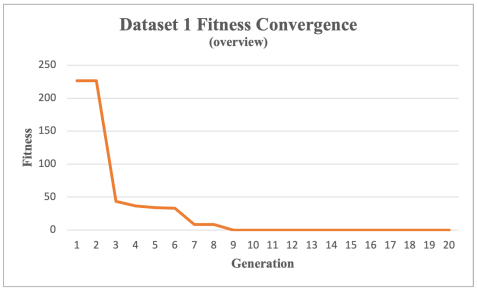

Figure 5: Dataset 1 Fitness Convergence

Below, Figure 6 zooms in to show the sudden drop in fitness between generations two and three. While we did see a steady decline in fitness after the third generation, it was almost always in the first three generations that a significant drop was seen. Largely this is likely because our crossover rate is very high, and hence we quickly create more complex trees that are far closer to the ideal expression tree for the data.

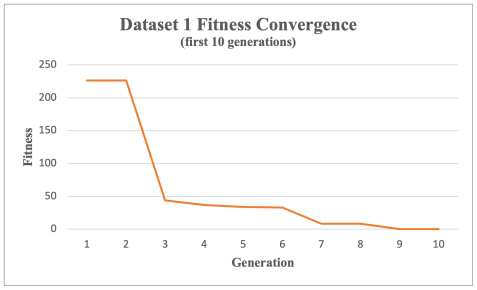

Figure 6: Dataset 1 Fitness Convergence within 10 Generations

Dataset 2

Upon running genetic programming on Dataset 2 by using the same set of parameters, mutation and crossover methods, and selection mechanisms as Dataset 1, our best-fit tree had a very high MSE of 3.54×10^8. On the test set, its MSE was 5.05×10^{20}, indicating that the equation was overfitting to the data, a phenomenon which we expected to occur. The change in fitness across generations in the best evolutionary run of Dataset 2 is shown in Figure 7.

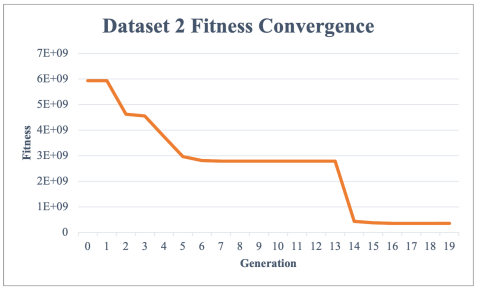

Figure 7: Dataset 2 Fitness over 20 Generations

In order to improve our results and combat the overfitting problem, we implemented a new mutation method and selection mechanism, using the methodologies described in the Experiments section above. After running genetic programming with both of these modifications in place, our best-performing tree simplified to the equation $-1089/(6.5321x_3)^2$ with an MSE of 1.05×10^9. On the test set, it had an MSE of 2.08×10^{12}. We see that while fitness during training increased slightly when compared to the previous set of evolutionary runs (MSE = 3.54×10^8), these modifications resulted in a lower fitness on the test set. Hence, although training fitnesses were higher than before, overfitting decreased, as seen through a relatively closer resemblance between our training and test set fitness values.

In all evolutionary runs of genetic programming on Dataset 2, there was a strong favoring of the variable x_3 in the equations of the best-performing trees of each generation. We were not entirely sure why this is, but hypothesize that it is a major contributing factor to the extremely high fitness values of these trees, and our inability to converge on a best-fit equation that includes all variables. One possible explanation is that the true best-fit equation of the data set contains x_3 with a high exponent, and in order to get to a higher order of the variable, our program favored it more heavily. Regardless, the favorable x_3 has a presence in all of our results, while equations with x_1 and x_2 were seldom selected. Overall, our Dataset 2 results were far less consistent and far worse than Dataset 1, despite our changes in mutation and selection methods, primarily due to the increased complexity of the data.

5 Conclusions

Our task to implement genetic programming to solve a symbolic regression problem was overall successful on Dataset 1, while yielding subprime results for Dataset 2. On Dataset 1 we were able to converge almost consistently, with the best result converging in nine generations, and our best-fit equation seemed to match the true equation well with a negligible MSE. This was $x^2 - 6x + 14$ with a mean squared error of 0.0006. Dataset 2 presented additional challenges with its three variables, and a more complex relationship between the variables. We modified our mutation and selection techniques, and were still unable to successfully converge at a

reasonably low value. Specifically our fittest fitness function was $-1089/(6.5321x_3)^2$ with a mean squared error of 1.05 x 10^9. Despite this, a point for further research could be isolating parameters and selection techniques in order to determine which specific implementations work better for such a data set. With more time, we would have been able to investigate this enquiry further, but perhaps that could be answered in a future paper.

6 Contributions

We were all able to offer input on every aspect of the assignment and complete the code as well as the paper during our in-person meetings. We each proof-read the entire document as well.

References

Champlin, R. 2018. Selection Methods of Genetic Algorithms. *Student Scholarship – Computer Science* 1–13.

Liang, Liu, X. 2020. Preference-driven Pareto front exploitation for bloat control in genetic programming. *Applied Soft Computing Journal* 1–18.

Silva, S., and Almeida, J. 2003. Dynamic Maximum Tree Depth. *Genetic and Evolutionary Computation — GECCO* 1–13.