Genetic Programming in the Symbolic Regression Domain

Brad Shook, Patrick Danielson and **Sebastian Charmot**
{brshook,padanielson,secharmot}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

In this paper, we implement a genetic algorithm in the symbolic regression domain and evaluate the accuracy of varying parameters on two different datasets. In particular, we investigate which combinations of parameters in our genetic program will generate the lowest mean squared error (MSE).

Our results indicate that our genetic algorithm on Dataset 1 performs best with parameter setting 8 whereas parameter setting 6 performed the worst (see Table 3 and Table 4). The main difference between these two parameter settings are the difference in population size, a larger population size resulted in better performance. For Dataset 2, our genetic algorithm did not converge on a function that correctly fit the dataset, however, the MSE of the functions did improve over generations (Figure 4).

1 Introduction

In this paper, our analysis is centered around creating a genetic algorithm that implements symbolic regression. Our goal is to create a genetic algorithm that can create functions that closely fit the data from our datasets.

A genetic algorithm has several general steps. It creates a random population and chooses the best individuals to reproduce and be passed on to the next generation. This process repeats with the expectation that the individuals of each generation will be more and more fit.

We followed many traditional genetic programming techniques, including crossover, mutation, and tournament selection (Oliveira et al. 2015). To ensure the algorithm did not create large individuals that over-fit our data, we implemented a death by size algorithm for bloat control. This process works by removing any trees that exceed a certain size (Luke and Panait 2006).

The genetic algorithm that we created has multiple parameters. We experimented with a variety of settings and gathered data to measure how effective each variation of parameters was. These experiments allow us to quantitatively evaluate which individual parameters had the most impact on the efficiency of the algorithm.

The rest of the paper will be structured in the following format. In the next section, background, we will describe the relevant terminology and parameters for our genetic algorithm. Then, we will describe how we evaluated our different parameters in the experiments section. We will continue by presenting the results of our experiments, and end on the conclusions that can be drawn from our results.

2 Background

Representation of Individuals

The first part of creating a genetic algorithm is determining a representation of the individuals in each population. In symbolic regression, this is typically done using trees, as seen in Figure 1 below. These trees have mathematical operators as non-terminal nodes and terminal nodes as either constants or input values to the function. Each tree can be solved recursively starting at the root node since each node below it is either a constant, input variable, or an operator.

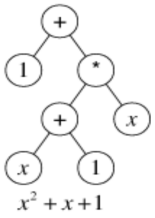

Figure 1: Example of an individual used in symbolic regression. (Koza)

Generating First Population

Each individual in the first population is created using random trees. To create a random tree, we first initialize a root node to be an operator. Then we add subsequent child nodes using a probabilistic method. We assign a probability that a subsequent node will be an operator or a non-operator.

- If it is an operator, there is an equal chance for any operator in the set $\{+, -, \times, \div\}$ to be chosen.
- If the subsequent node is a non-operator, we assign a chance for it to be either an input variable or a constant.

We add left and right children in the manner described above until neither children of a node are operators. Although our probabilistic method makes it unlikely for large trees to be created, we ensure this will not happen by using an initial maximum depth parameter that filters out any trees with a depth exceeding it.

Fitness

Each individual is assigned a fitness measure which determines how likely it is for that individual to crossover to the next generation. Each individual's fitness is the mean squared error (MSE) between the target value and its prediction for a random subset of 500 points from the training set. The random subset of 500 points is re-sampled every time an individual is assigned a fitness.

Tournaments

To select individuals for crossover, we run tournaments with a certain sample size from our current population and select two 'winners' from each tournament. We refer to the sample size of each tournament as the tournament size. Once we have randomly selected individuals for a tournament, we sort them by their fitness values such that fitter individuals are at the front of the list. A weighted probability list is then used to determine two winners for the tournament. This weighted probability list is weighted by the formula $p * (1 - p)^i$ where p is the starting tournament probability (see **Parameters**) and i is the index of the individual in the list. It is not always in the best interest to select the most fit individuals as this can decrease the diversity of the individuals in a population. Therefore, the goal of our weighted probability list is to select the better individuals most of the time. In theory this should allow us to balance an improvement in MSE and maintaining a diverse future population. The two winners of the tournament are then crossed over and added to the next generation's population.

Crossover

The goal of crossover is to create individuals with lower MSE values by combining two individuals from the previous generation. For our genetic algorithm, crossover is done by randomly choosing one node within each of the winning individuals from the tournament. We then switch the sub-trees at the randomly selected nodes to create two new children.

Mutations

We use mutations to increase diversity within each generation's population. A mutation could also generate enough noise to push an individual out of a local minima when the MSEs of the population are not improving. We perform a mutation on an individual by first finding a random node within the individual. We replace the subtree starting at the random node with a randomly generated tree. In our genetic algorithm, mutations can occur after performing crossover; there is a set parameter of probability (mutation probability in **Parameters**) which determines if the resulting individuals from crossover should be mutated.

Genetic Algorithm

Using the aforementioned processes, we have the elements necessary to create a genetic algorithm. Our genetic algorithm starts by generating an initial population and assigning a fitness value to each of the individuals within the population. From there, we run tournaments to find individuals to perform crossover on. After these new individuals are created, there is a chance a mutation will be applied to them. This process of generating new individuals is repeated until there is a new generation of individuals that is the same size as the last population (population size in **Parameters**). Our genetic algorithm runs for a set amount of generations but will stop if an individual meets a certain MSE threshold.

Parameters

The parameters that we use for our genetic algorithm are the number of generations, mutation probability, population size, initial maximum depth, maximum nodes, bounds for our constants, tournament size, starting tournament probability, and variable probability.

In order to adjust for bloat, we use a maximum nodes parameter to remove any members of the population with total nodes greater than maximum nodes. This is referred to as death by size and has been shown to be an effective method to control for bloat in the symbolic regression domain (Luke and Panait 2006).

The bounds for our constants refer to the range in which our constant values can be created. For example, if our range is between $[-5, 5]$, all of our constant values generated in the initial population will have a value in the range $[-5, 5]$. It is important to note that constant values outside that range can be artificially generated if two constants are the children of an operator node.

Our tournament size determines how many individuals we place into each tournament at a time. The starting tournament probability is used as the first weight in the weighted probability list in the tournament. This weight is then decreased by the formula shown in **Tournaments** to create the weighted probability list. The higher the starting tournament probability, the more likely the individual with the best fitness will be chosen.

The last parameter within our algorithm is variable probability. We use this parameter when generating nodes. When we generate nodes, there is a probability for whether a non-terminal or terminal node will be created. If a terminal

node is chosen to be created, there is then a probability for if a variable or constant node will be made. This probability for determining if a variable node will be chosen is the variable probability. This parameter is especially useful when trying to create individuals that are higher degree polynomials such as quadratics.

The specifics of how we selected the values for our parameters are explained in the experiments section.

3 Experiments

Our goal was to determine which combinations of parameters generate the best functions to model our two datasets. We did this by evaluating the MSE of the functions created by the genetic algorithm on testing data. Each dataset contained 25,000 points and was split into training and testing data, with 20% of the data being for testing.

We experimented with two datasets, each with inputs and outputs from two different functions. The first dataset, Dataset 1, had one input. The second dataset, Dataset 2, had three inputs. Early experiments on Dataset 1 indicated that we were under-fitting our data, so we increased our variable probability from 50% to 75% for its experiments. This allowed us to create functions of higher degree polynomials that fit Dataset 1 more closely.

	x	$f(x)$
mean	-0.0034	22.3594
stdev	2.8878	18.8824

Table 1: Dataset 1 Basic Statistics

	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
mean	4.9e+06	5.0e+06	5.0	6.8e+06
stdev	2.9e+06	2.9e+06	2.9	1.0e+09

Table 2: Dataset 2 Basic Statistics

After splitting the datasets, we created ten unique parameter settings for each dataset (see Table 3 and Table 4). Each parameter setting differs slightly so the effects of changes in parameters could be more easily identified.

Each combination of parameters was ran 20 times, and the results were averaged to reduce the influence of random chance on the results. Each instance of a genetic algorithm would either create generations until the most fit individual of a generation had an MSE less than .1 or 21 generations had been reached. For each generation created by the algorithm, the most fit individual's function, MSE value, number of nodes, and depth were collected. Collecting these attributes allowed us to analyze how the most fit individuals of each generation were improving.

4 Results

Dataset 1

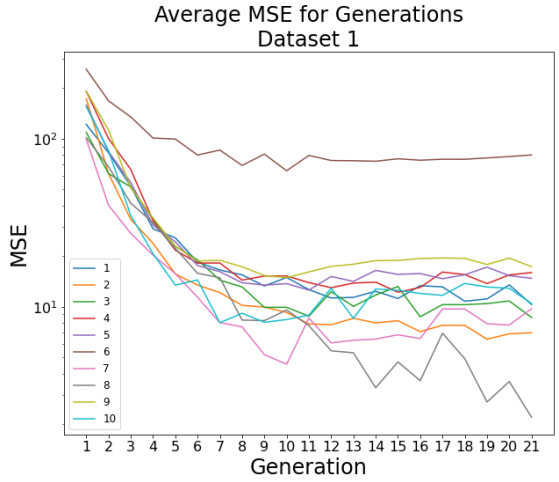

Figure 2: $\overline{MSE}$ vs. Generation for Dataset 1. A log scale is applied to the y-axis to better distinguish between lines.

The results for the experiments on Dataset 1 indicate that some parameter settings (PS) were more effective at fitting the data than others. As shown in Figure 2, PS_6 performed the worst in terms of average MSEs across generations. This is likely due to that fact that the population size for this setting was only 100, meaning there was less room for diversity among the first generation of individuals. The PS that performed the best were PS_8. We hypothesize this was the case because PS_8 had a population size of 1,000 and a constant range from -10 to 10. This meant that there was a large amount of diversity within the first and subsequent generations in terms of individuals and the constants within those individuals. Interestingly, changing the starting tournament probability did not have a noticeable effect on the algorithm. This can be seen in PS_4, the only PS which had a tournament probability which was not 0.8. PS_4 follows the same trend in average MSE as the rest of the parameter settings. Mutation probability did not have a noticeable effect on the average MSEs either. For example, PS_2 had a mutation probability of 30% and was one of the best parameter settings. However, parameter settings 3, 4, and 5 all performed worse in terms of average MSE. This could imply that increasing the tournament size for PS_2 was what improved its average MSE results.

Some parameters reached a reasonable MSE far more consistently than others. We define the number of times the algorithm correctly fit Dataset 1 as the number of times the final generation's most fit individual reached an MSE of .1 or below.

In Figure 3, we see that the parameter settings that caused the genetic algorithm to fit the data most consistently were PS_7 and PS_8. These two parameter settings fit the func-

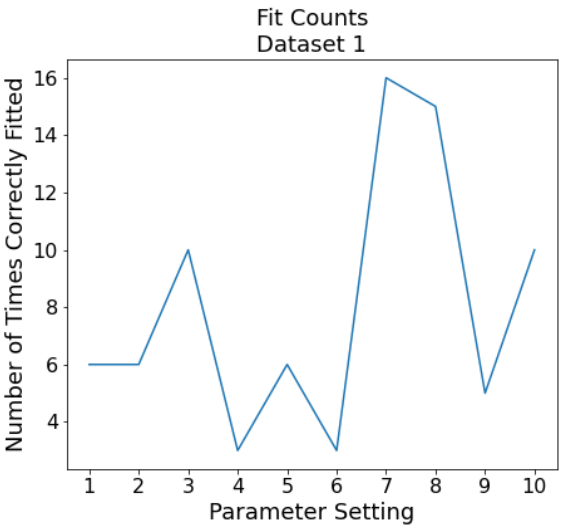

Figure 3: Comparison of how different parameter settings effected the number of times the genetic algorithm correctly fit the data with a MSE threshold less than 0.1.

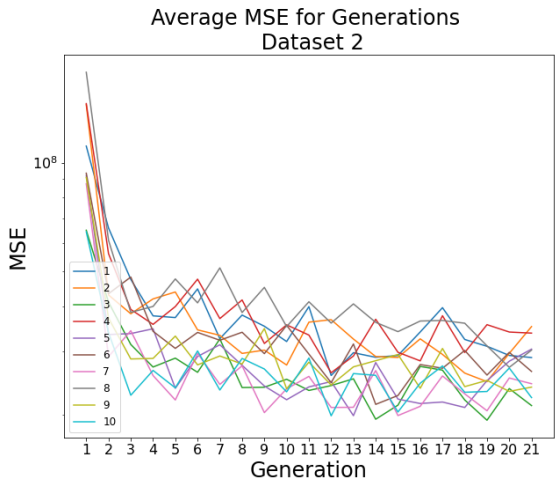

Figure 4: Comparison of Dataset 2's average mean squared errors across generations for all parameter settings. A log scale is used on the y-axis to distinguish between the different lines.

tions 16 and 15 times, respectively. This is likely due to PS_7 and PS_8 having the largest starting population size of 1000 among the parameters tested along with a low mutation probability. After analyzing Figure 2, it can be seen that those same parameter settings performed the best in terms of average MSE. Contrarily, PS_4 and PS_6 fit the data only two times each. PS_4 most likely performed poorly due to the starting tournament probability being 0.6 instead of 0.8. This probably caused less fit individuals to consistently make it to future generations. The reason PS_6 rarely fit the data can likely be attributed to its relatively small population size of 100.

Dataset 2

Our genetic algorithm failed to fit Dataset 2 as well as we would have hoped. Figure 4 shows that the average MSEs across generations, for all parameter settings, were in the millions. Despite not correctly fitting Dataset 2, our genetic algorithm did improve its MSEs across generations on average. As seen in Figure 4, the average MSEs in the early generations are quite high but decrease as new generations are created. The parameter settings that produced the best average MSEs were PS_3 and PS_7. This may have been due to the population size of 1,000, small constant range, and small variable probability. However, with there not being a large discrepancy between the average MSEs across the generations, it is hard to pinpoint the exact, or if any, parameters that improve the algorithm. Our analysis of the structure of the functions created by our genetic algorithm showed that the algorithm repeatedly converged on a function form of a/b where a and b were either constants or variables. The majority of these functions had a as $x3$ from Dataset 2. The values for this input were very large so it is likely the case that dividing this large value by smaller values created a baseline MSE. Our genetic algorithm seemed unable to remove itself from this local minima and converge on a smaller MSE. The cause of the large MSE values can be attributed in part to the scale of the data in Dataset 2. The standard deviation of Dataset 2 is 1.0e+09, which means that functions that deviate even slightly will have large MSE values. Even a few incorrect predictions could have a noticeable effect on the overall MSE.

5 Conclusions

In this paper, we were interested in how a genetic algorithm could be used to search for functions that could fit a dataset. We found that on a simple dataset, such as Dataset 1, our genetic algorithm performed strongly, especially with PS_8. However, on more complex, higher dimensional datasets, like Dataset 2, our genetic algorithm could not converge on a function that correctly fit the data. For both datasets, however, the size of the first population seemed to have the most impact on the success of our genetic algorithms. Our poorer performance for Dataset 2 indicates that a genetic algorithm is probably an approach that works best for datasets with relatively low variance.

Avenues for further investigation can be to recreate our genetic algorithm with modifications to our parameters. For example, the crossover function could be changed to reduce how different the children will be from its two parents. It might make sense to use a probabilistic method to encourage crossover on lower nodes of the parents in order to create children that are not too different from the parents. Additionally, an alternative bloat control technique could be used such as Double Tournament or Linear Parametric, both of which have been shown to be effective in the symbolic regression domain (Luke and Panait 2006). Lastly, it would

be interesting to test the algorithms with larger population sizes than in the paper. Due to computing constraints, the maximum population size tested was 1000. Since Dataset 2 contained such high variance, a much larger starting population size could induce enough randomness to generate the starting point of a converging function by chance.

6 Contributions

BS, SC, and PD all practiced paired programming for all coding. BS set up the individual representation and the generation of new individuals. SC was responsible for implementing the method for recursively solving individuals. PD worked on the mutation and crossover of trees, as well as tournament selection. SC combined the aforementioned processes to create the genetic algorithm. BS ran experiments and gathered results.

PD wrote the introduction and conclusion of our paper. SC wrote the background. BS wrote the experiments and results sections. BS also created the graphs while PD created the tables. All authors proof-read the entire document.

7 Acknowledgements

We would like to thank Dr. Raghu Ramanujan for insight and guidance throughout the process.

References

Koza, J. R. Example of a Run of Genetic Programming (Symbolic Regression of a Quadratic Polynomial).

Luke, S., and Panait, L. 2006. A comparison of bloat control methods for genetic programming. *Evolutionary Computation* 14(3).

Oliveira, L. O. V.; Otero, F. E.; Pappa, G. L.; and Albinati, J. 2015. Sequential symbolic regression with genetic programming. In *Genetic and Evolutionary Computation*. Springer International Publishing. 73–90.

Parameter Setting	Mutation Probability	Population Size	Initial Max Depth	Max Nodes	Constant Range	Tournament Size	Tournament Probability
1	10%	500	6	20	(-5,5)	8	0.8
2	30%	500	5	15	(-5,5)	12	0.8
3	20%	1000	5	15	(-10,10)	8	0.8
4	20%	500	5	15	(-5,5)	8	0.6
5	30%	500	5	15	(-5,5)	8	0.8
6	10%	100	6	20	(-5,5)	8	0.8
7	10%	1000	6	20	(-5,5)	8	0.8
8	10%	1000	6	20	(-10,10)	8	0.8
9	10%	500	5	15	(-5,5)	8	0.8
10	10%	500	6	20	(-5,5)	12	0.8

Table 3: Dataset 1's parameter settings for experiments.

Parameter Setting	Mutation Probability	Population Size	Initial Max Depth	Max Nodes	Constant Range	Tournament Size	Tournament Probability	Variable Probability
1	10%	500	12	30	(-2,2)	12	0.8	75%
2	20%	500	12	30	(-2,2)	12	0.8	75%
3	30%	1000	12	30	(-10,10)	12	0.8	50%
4	30%	500	12	30	(-500,500)	12	0.8	50%
5	40%	1000	8	25	(-13,13)	10	0.8	75%
6	15%	700	10	30	(-25,25)	12	0.8	20%
7	23%	1000	23	17	(-39,39)	10	0.8	20%
8	30%	500	12	30	(-500,500)	16	0.6	50%
9	30%	1000	12	30	(-500,500)	16	0.6	50%
10	30%	1000	12	30	(-1000,1000)	16	0.8	50%

Table 4: Dataset 2's parameter settings for experiments.