# Symbolic Regression Using Genetic Programming

**Chris Lamb** and **James Ni**
{chlamb,jani}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

**Abstract**

Curve fitting is a common requirement for data analysis. It is valuable in many disciplines to be able to construct a function that predicts future values based on past data. We preformed symbolic regression using genetic programming on two data sets with the goal of determining the functions used to generate the data sets. Our implementation represented functions as expression trees with operators, variables, and coefficients as the nodes of the trees. By manipulating various parameters in the genetic algorithm, we were able to perform fits to both data sets and find simple representative expressions. In the process, we explored methods of dealing with the common pitfalls of genetic programming bloat, over-fitting, and loss of diversity.

## 1 Introduction

Finding trends is an important part of transforming raw data into usable information. It is challenging to take large amounts of data and find an overarching trend without computational assistance to evaluate the data. Thus curve fitting is an important topic in the field of computer science, which is largely dedicated to intelligently handling data. One common method of performing this task is symbolic regression using genetic programming.

We investigated symbolic regression using genetic programming and explored its benefits and potential pitfalls. We attempted to determine the functions used to generate two large sets of data. For each experimental run, we started with an initial population of randomly generated expressions and combined them based on their fitness to model the data sets. We repeated this process for many generations. We experimented with tuning a number of parameters such as the population size, mutation rate, and method of selection for crossover to optimize our genetic algorithm and the functions it produces.

This paper is organized as follows: in Section 2, we discuss the details of genetic programming and symbolic regression. In Section 3, we explain our experimental method. In Section 4, we present our results. In section 5 we conclude by discussing the quality of the functions our algorithms generated and what our experiment demonstrates about the nature of genetic programming. Finally, in section 6 we detail the individual contributions to this research.

## 2 Background

### Genetic Programming

Evolution due to natural selection is an incredibly powerful force in nature that enhances a populations fitness generation by generation over an extended period of time. Computer scientists have adapted this phenomenon in the form of genetic programming and found it extremely successful at solving problems in certain domains where other common methods are ineffective. Genetic programming works much like natural selection by randomly generating an initial population and future generations being spawned by combining the information of the fittest individuals. This process can be repeated until an individual of the desired fitness is created. Genetic programming seems quite simple but there are a number of obstacles that must be overcome in order for it to be successful.

Genetic programming requires a good fitness function that accurately assesses how close a particular individual is to a desired solution. Genetic diversity must be maintained in order for progress to be made. Finding solutions that are local maximums on the fitness spectra can wipe out diversity if counter measures are not taken. As genetic programs progress the individuals in their population can become bloated using long convoluted representations of simple ideas and this can cause the program to slow down before a desired solution is found. Another pitfall of genetic programs is over-fitting. This means coming up with a complex solution that perfectly fits the data that the program learned on but that is not predictive of future data points. In the following section we will discuss how we attempted to deal with each of these issues.

### Symblolic Regression

The particular application that we used genetic programming for was symbolic regression. Symbolic regression is a method of determining a function that fits a set of data. The function or expression is repressed in a tree structure called an expression tree. The internal nodes of the expression tree contain operators and the terminal nodes of the tree contain

coefficients or variables. The tree is evaluated by a post order traversal of the tree with operators being applied to their children. This representation lends itself to genetic programming with trees being able to exchange sub-trees with each other to generate offspring. This method of regression is robust as it doest not assume anything about the form of the final function.

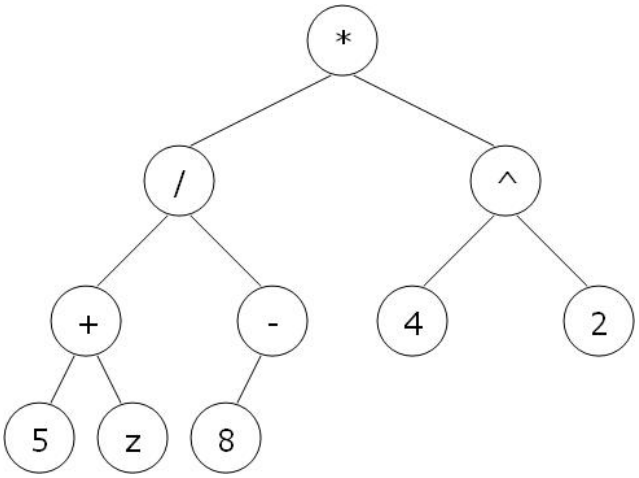


Figure 1: Illustration of a binary algebraic expression tree (Wikipedia 2017)

## 3 Experiments

We set out to implement symbolic regression to determine the functions that created two sets of data. Each set was generated using exclusively addition, subtraction, multiplication, and division so $+, -, *, \div$ made up our operator set. Our coefficients consisted of only integers for the first data set and spanned all real numbers for the second data set. There was only one variable $x$ for the first data set and there were three variables, $x_1, x_2, x_3$, for the second data set.

We constructed the expression trees that constituted our initial population by supplying a depth parameter $d$ and then generating a list of $2^d - 1$ random operator nodes, a list of variable nodes with each variable in the data set, and $2^d - n$, where $n$ is the number of variable nodes with random coefficients. We ran each data set with different population sizes but settled on 10000 for the first data set and 1000 for the second data set as a good balance between running through generations quickly and maintaining diversity. Our fitness function consisted of the root-mean-squared error over all of the points in the data set. We also penalized the fitness of expressions that attempted to preform the illegal operation of dividing by zero by setting their fitness to $\infty$ preventing them from being selected for crossover. We choose individuals for crossover using their fitness as a proportion of the total fitness of the entire population and we did not allow individuals to crossover with themselves.

Using this method we selected two trees at a time and randomly choose an edge from each tree to swap the two swapped trees became a part of the next generation. We repeated this process until the new generation reached the population size. Each newly spawned offspring was then mutated and simplified. We initially tried a mutation rate of 2% based on it's success for Miller and Thompson (Miller and Thomson 2000) however we found that 5% gave us better results. During mutation, each coefficient had a 5% probability of being changed. Initially we did not change operators or variables through mutation as those changes are more drastic and we wanted the drastic changes to happen through fitness selection and not random chance. However, after some testing we found that a second type of mutation where a randomly selected node from the tree was replaced with a new randomly generated tree with a 5% probability helped maintain a diverse population. We also algebraically simplified each expression tree combining coefficients and collapsing variables to help eliminate bloat. (Wong and Zhang 2006) In addition to the crossed over offspring we also allowed the top .1% of the previous generation to survive. We initially set the maximum number of generations to 200 per run and also created a cut off value and terminated the program if the fitness of the fittest individual was better than the cut off. However, for the second data set, we noticed that the root-mean-squared error was continuing to decrease at 200 generations so we removed the maximum number of generations parameter to avoid ending the regression too early.

After a number of unsuccessful runs with data set two we examined the data and noticed that $f(x)$ appeared to have an inverse relation to $x_3$ indicating that the correct function would have an $x_3$ term in the denominator however some of the values of $x_3$ were 0 we assume due to rounding which led our algorithm to throw out expressions that has $x_3$ in the denominator. We decided to remove all the points where $x_3$ was zero from the data set and our results improved significantly.

We implemented a maximum tree size and threw out any trees that exceeded the maximum number of nodes to combat over-fitting and bloat. After some experimentation we settled of 60 nodes for this number. This prevented expression trees from becoming overly complicated.

Finally we partitioned each data set into a training set and a testing set allowing our algorithm to learn from the training set then be evaluated based on its performance on the testing set. This was done in another attempt to prevent over-fitting. We divided our data equally between training and testing.

## 4 Results

We ran our program using the first data set with an initial population of 10000 trees, a cross generation survival rate of .1%, and integer coefficients between the values of $-5$ and 5 for the initial population. The mutation rate was 2% and the minimum and maximum changes for each coefficient mutation were $-3$ and 3 respectively. Using these parameters,

an expression with fitness lower than our cut-off fitness of .1 root-mean-squared typically appeared within three to six generations. The program generated the expression

$$f(x) = x^2 - 6x + 14 \tag{1}$$

which we believe to be the function used to generate the data set. The root-mean-squared error for this function over the first data set is 0.0245 which can be attributed to the data set having only two decimals of precision for any one value. The $R^2$ value for this function is 0.999. However, occasionally the program reached a linear function that fit the data sufficiently well that it dominated the proportional crossover and prevented the algorithm from ever reaching the optimal quadratic solution. This demonstrates the importance of resets in genetic programming in order to avoid getting stuck at local minimums or maximums.

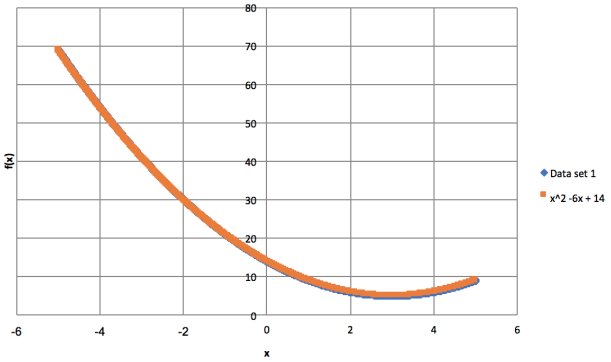


Figure 2: Plot of data set 1 overlaid with out best fitting result

The second data set proved significantly more challenging, but ultimately we did find a function that we believe generated the data set. The best fitting result we achieved had a root-mean-squared error of 18599.857 on the training set. It was obtained with a survival rate of of 0.1%, a population size of 1000 trees of initial depth 3, initial real coefficients between $-1$ and 1, and coefficient mutations between $-1000$ and 1000. The simplified best-fitting equation was

$$f(x_1, x_2, x_3) = 2.79447 \times 10^{-9} \frac{x_1 x_2}{x_3^2}. \tag{2}$$

The root-mean-squared error of this function on the full data set is 5483330.446 and the $R^2$ value is .975.

This equation appeared to have the same functional form as inverse square forces, such as the force due to gravity, which has a constant of a similar order of magnitude. The equation for gravity is thus

$$f(x_1, x_2, x_3) = G \frac{x_1 x_2}{x_3^2} \tag{3}$$

where the gravitational constant $G = 6.67 \times 10^{-11}$. This gave a root-mean-square error of 30200 and an $R^2$ value of .951. While the $R^2$ value is lower than that of the equation generated by the program, we believe that can be attributed to all of the values being rounded down. The root-mean-squared value of the gravity equation signifies that it is a much better approximation at the high values of $f(x)$ when $x_3$ is small. The large errors we still see can be attributed to the fact that the values given in the data sets are all truncated to two decimal places of precision.

## 5 Conclusions

In conclusion, our genetic program was quite successful in determining the function used to generate the first data set and had sufficient success approximating the function used to generate the second data set that we were able to deduce it. This exercise demonstrated the potential benefits of genetic programming, successfully fitting both data sets, but it also demonstrated the pitfalls. The extreme sensitivity to initial conditions can lead to very different outcomes for the quality of the results found by genetic algorithms and requires a great deal of meticulous testing. Genetic programming also depends heavily on random chance, as it is possible to get stuck on linear solutions and other local maximums which can destroy diversity and require restarts.

## 6 Contributions

CL implemented the infrastructure code for representing expression trees, crossover, mutation, and simplification. JN implemented reading in and storing the data sets, simplification, sub-tree mutation, crossover selection, and complexity limitation. Both JN and CL contributed to researching and testing parameters for the symbolic regression. Both CL and JN contributed to every section of the paper.

## References

Miller, J. F., and Thomson, P. 2000. Cartesian genetic programming. In *European Conference on Genetic Programming*, 121–132. Springer.

Wikipedia. 2017. Wikipedia. `https://en.wikipedia.org/wiki/Binary_expression_tree`. Retrieved on Feb. 15, 2017.

Wong, P., and Zhang, M. 2006. Algebraic simplification of gp programs during evolution. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 927–934. ACM.