Symbolic Regression with Genetic Algorithm

Richard Hu and **Daniel Zhou**
{rihu,dazhou}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

This research paper explores the application of Genetic Algorithms (GAs) in symbolic regression, with a focus on three distinct data sets. The datasets vary in complexity, from single-variable datasets to those featuring multivariate, trigonometric, logarithmic, and exponential functions. In three distinct datasets, Genetic Algorithms (GAs) were employed for symbolic regression with different data engineering techniques and hyper parameters. The results demonstrated that the GA-based approach successfully discovered accurate mathematical expressions for each dataset, while the convergence rate for each varies.

1 Introduction

The objective of this paper is to investigate the application of genetic algorithms in symbolic regression. Symbolic regression aims to find a mathematical formula that best represents a given dataset. Unlike conventional regression techniques, which fit data to a pre-defined model (Pedace 2013), symbolic regression searches for the most fitting model within a set of possible mathematical expressions (Koza 1990). This is an important area of study as it allows for a more flexible and interpretable model, which is particularly useful in fields such as scientific research, financial modeling, and medical diagnostics (La Cava et al. 2023).

The scope of this study is confined to the implementation and evaluation of symbolic regression using genetic algorithms for three specific types of datasets:

- single-variable datasets,
- sulti-variable datasets, and
- single-variable datasets featuring trigonometric, logarithmic, and exponential functions.

Our approach involves representing mathematical expressions as tree-like data structures, where each node in the tree is an operation, variable, or constant. We then apply genetic algorithms to evolve these expressions over successive generations, aiming to minimize the difference between the predicted and actual values in the dataset. The genetic algorithm utilizes crossover, mutation, migration, and selection operations tailored for our tree-based representation of mathematical expressions.

The rest of this paper is organized as follows. The background section provides the background knowledge for our approach to genetic algorithms including how we process data, and how we handle mutation, crossover, and selection. The experiment section describes the detailed implementation and experimentation on three different types of datasets. The result section discusses the findings, evaluates the performance of the algorithm, and compares the results using different parameters. The conclusion section concludes the key findings and off-sight.

2 Background

Symbolic Regression and Genetic Programming are potent techniques. This section defines and outlines key components of our implementation and provides the background knowledge necessary for understanding the algorithms.

Symbolic Regression

Symbolic Regression is a type of regression analysis focused on identifying mathematical expressions that best fit a given dataset. Unlike traditional regression methods, which employ predefined functional forms (Pedace 2013), symbolic regression offers greater flexibility by enabling the discovery of novel mathematical relationships within the data (Koza 1990). This approach is particularly valuable in fields such as scientific research, finance, and healthcare, where understanding the underlying mathematical relationships is crucial (La Cava et al. 2023).

Genetic Algorithm

Genetic Algorithms (GAs) are optimization methods inspired by natural selection and genetics. These algorithms are commonly used to find approximate solutions for search and optimization problems (Katoch, Chauhan, and Kumar 2021). In our implementation, the population consists of a pool of randomly generated functions $f(x)$, each having a given fitness. These functions are evolved over generations to improve their fitness, based on a defined fitness function. After generating the children's population through mutation and crossover, we select the new generation according to their fitness, giving better-fit solutions a higher chance of moving on to the next generation. The process of evolution continues until the convergence condition is met.

Expression Tree

In our approach, mathematical expressions are represented using expression trees. Each node in the tree can be an operation (e.g., addition, multiplication), a variable, or a constant. This tree structure naturally accommodates nested mathematical expressions and simplifies the implementation of genetic operations like mutation and crossover. Expression trees can be traversed in-order, with the terminal nodes, either variables or constants, serving as the leaves of the tree. The tree is built using a depth-first-search approach where each node has the same probability for being any terminals and operators. To prevent excessively large trees, we restrict the number of operators in an expression tree.

Fitness Function

The fitness function serves as the evaluation metric for candidate solutions in the genetic algorithm. In our implementation, we use two kinds of function: mean square error (MSE) and percentage mean square error (PMSE), which will be defined in the experiment section. To penalize overly complex expression trees, we adjust the fitness function by multiplying it by the logarithm of the number of operators used.

Mutation

Mutation introduces genetic diversity into the population by altering a randomly chosen node in an expression tree. In our implementation, we use two kinds of mutations: switching a node between operator or terminal (variable or constant), and generating a random subtree from a node. To restrict the influence on the whole population, mutation is introduced with a low probability.

Crossover

Crossover, or recombination, combines genetic material from two parent solutions to create one or more offspring. In the context of symbolic regression with expression trees, subtree crossover is often employed. In this method, subtrees from two parent trees are exchanged to create new offspring trees. Subtrees from each parent are randomly selected for swapping.

Migration

We enhance diversity by incorporating randomly generated expression trees into the population, particularly in multivariate symbolic regression. After generating the children's population through crossover and mutation, a low probability is set for introducing new randomly generated expressions into the new population. Migration also occurs when an expression in the current population has too many operators–it will be "killed" and a new expression is generated to replace it.

Selection

Selection is the process by which individuals are chosen to serve as parents for the next generation. Various techniques for selection exist, including roulette wheel selection, tournament selection, and elitism (Blickle and Thiele 1996). Our algorithm employs roulette wheel selection, whereby a random number is generated each time an expression is selected from the new population. Different probabilities are assigned to best-fit children, parents, less-fit children and parents, and random new expressions. The sum of these probabilities is one. The newly selected population maintains the same size as the previous generation but is expected to have an improved overall fitness.

3 Experiments

In order to gain an in-depth understanding of the GP algorithm, we implemented on three different datasets, each with 25000 data points. These datasets have different features which lead to different implementation of the GP algorithm.

GP for Dataset1

The f function to be guessed from dataset1 is composed only of $+, -, \times, \div$, and integer constants. By studying the dataset before implementing GP, we observe that f is quite smooth, i.e. there is no big fluctuation or significant outliers. The ranges of x and $f(x)$ in the dataset are also relative small: $-5 \le x \le 5$ and $5 \le f(x) \le 68.99$. Intuitively, we think f is not very complicated and the basic implementation of GP could guess it, without special techniques. The only thing that may cause worry is that each fixed value x corresponds to several different $f(x)$ in the dataset. According to the author of this dataset, this error is caused by the rounding of input and output. The consequence of this error is that, unless the fitness function is defined in a very special way, we could never found a function with fitness 0, since no valid functions allow multiple output values at the same input.

For simplicity of notation, denote the input and output in dataset1 be $x_1, x_2, \ldots, x_{25000}$ and $y_1, y_2, \ldots, y_{25000}$, where y_i is the rounded $f(x_i)$. Let f^* be a candidate function. Define the number of operators in f^* as $depth(f^*)$. To overcome the over fitting problem, we set the first 10000 data points as training set and the rest 15000 data points as the test set. We didn't define a selection function for the training set because the data points in dataset1 are random distributed by inspection. Then, we define our fitness function, denoted by h, as

$$h(f^*) = \frac{1}{10000} \sum_{i=1}^{10000} (f^*(x_i) - y_i)^2 \cdot \log(\log(depth(f^*) + 2) + 3).$$

This fitness function is consisted of two parts. The first part, $\frac{1}{10000} \sum_{i=1}^{10000} (f^*(x_i) - y_i)^2$, is the general MSE. The second part, $\log(\log(depth(f^*) + 2) + 3)$, is the regularization term used to "penalize" candidate functions with high complexity (large number of operators). The two log operators in the regularization is used to restrict this penalty so that complex functions will not be dropped completely. The constant 2 and 3 in the regularization term guarantees that domain errors do not occur.

Similar to the fitness function, we also define the evaluation function, denoted by g, as

$$g(f^*) = \frac{1}{15000} \sum_{i=10001}^{25000} (f^*(x_i) - y_i)^2 \cdot \log(\log(depth(f^*) + 2) + 3).$$

This function is used to evaluate the candidate function after GP generated it from the trainning set.

Each expression can be represented as a binary tree whose nodes are chosen from $\{1, 2, 3, 4, x, +, \div, \times, -\}$. The seed population is generated by regression, i.e. we first assign value to the root nodes, then generate its left and right child. When doing regression, we keep track the number of operators used, and we restrict each expression in the seed population to use at most 4 operators.

The size of the population of each geneation is set to be 200. Suppose the current population is P. To generate the next generation, we first do crossover operations among P to generate a children population, denoted by C, with size 400. Each term of C is mutated with probability 0.2. Next, P and C are sorted with respect to the fitness of each term. Finally, we generate the next generation with population size 200 by adding terms to it one by one. Each term has

- 0.2 probability to be the expression with smallest fitness from P that has not been added before,
- 0.1 probability to be the expression with largest fitness from C that has not been added before, and
- 0.7 probability to be the expression with smallest fitness from C that has not been added before.

Note that we also add expressions with large fitness in order to preserve the diversity of each population, in the simple logic that bad parents can also generate very good child.

The crossover operation is defined as randomly choosing two nodes from two expressions and exchange the subtrees induced by them.

The mutate operation is defined as changing a random node that is assigned x or a constant. If it's asigned a constant, we change it to x; if it's x, we change it to a random constant chosen from $\{1, 2, 3, 4\}$.

Each evolution run will be terminated in one of the two ways: either the expression in the current generation has fitness less than 0.001, or the smallest fitness in all generations has not been updated for 10 generations. After a run is terminated, the expression in all generations with the smallest fitness will be returned as a candidate funtion. We performed 10 independent evolutionary runs and chose the candidate function with the smallest evaluation value as the best-performing model.

It should be noted that when calculating the fitness of a function, we may met the problem of something divided by 0, which is not allowed arithmetically. When we encounter such problem, we simply change the subtree induced by that $\div$ into 0. Since we only evaluate the fitness when we generate an expression, which will be kept as an atribute, this change won't modify the expression tree once it's generated and our expression class is still immutable.

The parameters chosen for the implemention of GP, especially on dataset1, are made arbitrarily with intuition. Indeed, dataset1 is relatively simply and our first version of implementation is good enough to find a very good guess function. Since the dataset has multiple values for the same input, our function could not match each data point. But upon careful inspection, for each input, we think our guess function approximately lies on the mean of the corresponding outputs in the dataset.

GP for Dataset2

Similar to dataset1, the f function in dataset2 also uses only four kinds of operators ($\{+, -, \div, \times\}$). However, it uses real constaints and has three input variables (x_1, x_2, x_3). Moreover, the ranges of input and output in dataset2 is relatively large:

$$\begin{aligned} 98 \leq & x_1 \leq 9999629, \\ 188 \leq & x_2 \leq 9999545, \\ 0 \leq & x_3 \leq 10, \\ 0 \leq & f(x_1, x_2, x_3) \leq 160629674968.55. \end{aligned}$$

These properties of dataset2 makes it relatively difficult, for GP especially, to find an appropriate candidate function.

However, with careful inspection and implementation, we still managed to let GP find a very good guess for f. To summarize, there are five key difference between the implementation for dataset2 and the implementation in dataset1. We will focus on these differences in this section.

First, we notice that the mutation operation is hard to generate huge constant. When we study dataset2, we tried to calculate an appromixate function by hand. Actually doing calculation by hand is much more efficient than running GP on computer and we found a good candidate in 10 minutes. The big discovery is that this candidate has a term of $\div 10^{12}$. However, it's almost impossible for GP to generate a constant as large as 10^{12}. (It can be factored, of course, but no matter how to factor it, the corresponding terms are still very hard to generate.) With advise from Dr. R, we normalized the input data. We found that the term of 10^{12} is almost completely compensated after normalizing the input and our hand-approximate candidate can be expressed using constans less than 100, a reasonable range for GP.

Second, the large range of output turns out to be a trouble in an indirect way. Not only the value of these output are not uniformly distributed, but the density in different ranges differs in a drastic manner. In this case, MSE may not be a good choice for fitness function. Instead, we use percentage mean square error (PMSE) as the fitness function, defined as following:

$$h(f^*) = \frac{1}{500} \sum_{i=1}^{500} 10000 \cdot (f^*(x_i) \div y_i - 1)^2 \cdot \log(depth(f^*) + 8)$$

where each x_i represents a set of choice of x_1, x_2, x_3, $\frac{1}{500} \sum_{i=1}^{500} 10000 \cdot (f^*(x_i) \div y_i - 1)^2$ is the PMSE term, and $\log(depth(f^*) + 8)$ is the regularization term. Notice that we change the training set size from 10000 to 500. This is because the complexity of dataset2 requires more runs to give good candidates and we have to enhance the speed of generating a new generation. Also, to enhance the performance of GP, we increase the population size to 350 and the maximum number of operators allowed in seed generation to 5. Since the training set size is so small, we set the test set to be the whole dataset. The selection function for new generation is also changed: with 0.5 probability we add the expression

from the parent generation with smallest fitness that has not been added before, and with 0.5 probability we add the expression from the child population with smallest fitness that has not been added before. We also preserves the diversity: when parents are chosen to perform crossover operation, if a parent uses more than 50 operators, we will generate a new expression to replace it (migration). This change also helps to prevent candidate function to be too complicated. In fact, if we don't set a bound for the number of operators, python could hit its regression limit very quick. Another way to avoid it is to add a very strong penalty term for regularization, but we worry that such a term could weaken the evaluation ability of the fitness function.

Third, upon careful inspection, we found f is very sensitive to x_3 when it's small, an implication that m_3 may appear in a dividing term. This could also cause trouble because the input in dataset2 are rounded to the two decimal place. Therefore, when x_3 is very small, $f(x_1, x_2, x_3)$ may be very far away from y, the approximate term of $f(x_1, x_2, x_3)$ in the dataset. This is an inherent defect of the raw data in dataset2. Since we could not find the exact input and output, we need to delete those data points with very small x_3. Also, since we are using PMSE instead of general MSE, when y is very small, the rounding error may cause huge difference. For instance, if $f(x_1, x_2, x_3) = 0.003$ and y is rounded to 0, then there is a 100% error of y. Thus we also need to delete the data points with very small y. Since every y in the dataset is non-negative, this change deletes all the data points where $y = 0$. Thus we will not encounter divide 0 error when calculating PMSE.

Fourth, we observe that, after deleting these extreme points, the smallest fitness converges much slower from generation to generation. In other word, GP could perform better if it contains some extreme points. Why? Well, as we stated before, the output in dataset2 is very different from uniform distribution and there are relatively few points with very large or small output. Thus, these extreme points actually illustrate more properties of f than general points do. Based on this key observation, we make a careful choice of the 500 data points in the training set:

- Delete data points whose y or x_3 is less than 0.1 (368 points get deleted). The rest data points are the test set.
- Pick 100 data points from the 500 dada points with lowest y. Put these 100 points into the training set.
- Pick 100 data points from the 500 dada points with highest y. Put these 100 points into the training set.
- Pick 300 data points from the rest of the dataset. Put these 300 points into the training set.

This special choice of training set greatly enhanced the performance of GP.

Fifth, when implementing GP for dataset1, we simply delete the whole subtree when devide 0 error is encountered in calculating fitness. This is not the best choice, as we lose a lot of information when cutting off the whole subtree. When implementing GP for dataset2, we don't delete the whole subtree, but modify the expression a little. Suppose the problem we met is $\frac{p}{0}$ where p is the expression of the induced subtree. We will now change it into

$$\frac{p}{0 + 0.0000000001}$$

and then do the calculation. Readers may wonder why don't just return huge number of this dividing operation so the expression will have a huge fitness and will not be chosen. This seems to work, but it has a disastrous defect in logic: the fitness of that wrong function could be very small, even if the dividing operation returns a huge value. Suppose

$$p^* = f + \frac{1}{\frac{p}{0}}.$$

If we set $\frac{p}{0}$ as infinity, then the output of p^* will be almost the same as f, forcing its fitness to be very small and p^* could be chosen as a candidate. But p^* should not be chosen, as dividing 0 is not allowed arithmetically.

There are other small changes. For example, we don't generate constant from $\{1, 2, 3\}$, but as a random real number from 0 to 10. Since this function is not simply, we do not have clear expection of the time needed for an independent run before we actually do the test. Thus, instead of fixing the number of independent runs, we setup a time limit of 7 hours for the machine (all experiments are performed on Intel(R) Xeon(R) Gold 6226R CPU running at 2.9Ghz with 348G of RAM). In other word, no more runs will be attempted after GP runs for 7 hours. This change also helps us evaluate the average time needed for each independent run. The mutate operation is now defined as choosing a random node and changing it to a random generated subtree. This mutation performs better than the simply mutation operation defined in previous subsection. Since PMSE is harder to optimize when it's very small, we terminate a run when the smallest fitness is less than 0.1 instead of 0.001. We also terminate it if the smallest fitness has not been changed for 20 generations, instead of 10. Based on systematic testing, we setup another way to terminate an evolutionary run: if there has been 50 generations and the smallest fitness is at least 30, or if there has been 100 generations. We terminate in this way because it's usually much slower to process later generations than earlier generations, and that most runs that converge well has a smallest fitness under 30 in 50 generations.

GP for Dataset3

This dataset is very similar to dataset1, except that four additional operators are allowed to use: $\{\log, \exp, \sin, \cos\}$. The range of the input and output in dataset3 is

$$-5 \le x \le 4.999,$$
$$-2917.159 \le f(x) \le 3729.416.$$

Similar to dataset1, we will change the subtree to 0 when encountering divide 0 error. When evaluating exp of some number larger than 30, we will return 1000000000. This is to avoid overflow problem and the quality of the candidate can be checked later with a graph. Also, when cos or sin is evaluated upon a value whose absolute value is larger than 1000000000, we will also change the subtree to 0. This

helps avoid a strange domain mistake in python. The fitness and evaluation functions are defined as

$$h(f^*) = \frac{1}{500}\sum_{i=1}^{500}(f^*(x_i) - y_i)^2 \cdot \log(depth(f^*) + 8),$$

$$g(f^*) = \frac{501}{25000}\sum_{i=501}^{25000}(f^*(x_i) - y_i)^2 \cdot \log(depth(f^*) + 8).$$

The regularization term is $\log(depth(f^*) + 8)$. Seed generation is generated in the same way as for dataset1. Population size is set to be 350 and the maximum number of operators in seed operation is 5. Mutation is defined similar to the implementation for dataset2, where a random node is replaced with a new subtree. Constants that can be chosen as node value is now $\{1, 2, 3, 4, 5\}$. The mutate rate is still 0.2. The number of independent evolutionary runs is 300. With systematic testing, we terminate each run in any of the following case:

- The smallest fitness has not been updated for 20 generations;
- The smallest fitness is less than 1;
- There has been 50 generations but the smallest fitness is at least 10000000;
- There has been 80 generations but the smallest fitness is at least 50000.

Similar as for dataset2, a parent is replaced with a new expression if it uses more than 80 operators. Similar as for dataset1, we generate the next generation with population size 350 by adding terms to it one by one. Each term has

- 0.15 probability to be the expression with smallest fitness from P that has not been added before;
- 0.1 probability to be the expression with largest fitness from C that has not been added before;
- 0.1 probability to be a new expression (migration);
- 0.6 probability to be the expression with smallest fitness from C that has not been added before.

4 Results

GP for Dataset1

The best candidate found by GP for this dataset, denoted by f^*, is

$$f^* = ((2 + (x * ((x - 2) - 4))) + (2 * (2 + 4))),$$

which can be simplified into

$$f^* = (x - 6)x + 14.$$

As specified in the experiment section, this is the candidate with smallest fitness from the 10 independent evolutionary runs. These 10 runs take 3692.5 seconds in total. We observe that in most of these funs, the smallest fitness falls under 2 in 10 generations, and f^* only uses 7 operators. This reflects that the function of dataset1 is not complex and can be easily approximated with GP. Using h and g, the fitness and evaluation functions defined in the experiment section with rounding to 5th digit after the decimal, we have

$$h(f^*) = 0.00035,$$
$$g(f^*) = 0.00035.$$

Notice that h and g contain the regularization term to punish functions with high complexity. If we drop the regularization term, we see that the MSE of f^* on the training set (first 10000 data points) is 0.00059 and the MSE of f^* on the test set (last 15000 data points) is 0.00059. These data already shows f^* is a good guess.

Since this dataset has only one input, we can visualize f^* and the dataset with a general 2-dimensional graph, as shown in Figure 1. From this picture, it's clear that f^* is a good guess. If we zoom in , we would see that on each input value, f^* approximately lies on the mean of the multiple output values in the dataset (which is due to the rounding error of the dataset).

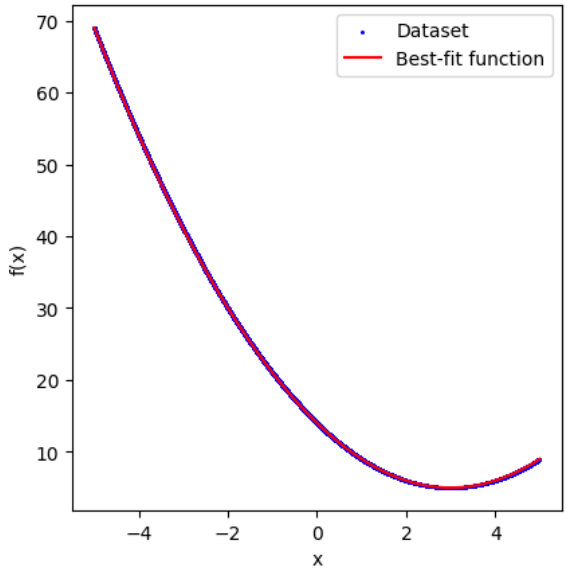

Figure 1: Plotting of best candidate f^* for dataset1.

Besides the set of parameters stated in the experiment section, we also play around with other choices of parameters of population size, number of independent evolutionary runs, max number of operators allowed in seed population, and constants that can be chosen to assign to a node. In general, we observe that as population size increases, the smallest fitness converges with a faster speed, but it also takes longer to generate a new generation. Since this function is so simple, any set of parameters we chose could give a good guess in the end.

GP for Dataset2

The best candidate found by GP for this dataset, denoted by f^*, is

$$\begin{aligned} f^* =& ((((((x_2 - 188.0)/9999357.0) * 8.855973601356151) \\ & /0.13733814259975774) - ((0.13733814259975774 \\ & - (8.214452763286603 - 5.861747531267377)) \\ & * ((x_2 - 188.0)/9999357.0))) * ((((x_1 - 98.0)/9999531.0) \\ & /((x_3 - 0.0)/10.0))/((x_3 - 0.0)/10.0))). \end{aligned}$$

We did not found a valid tool to simplify it because it's too long. As specified in the experiment section, we setup the time limit of GP to around 7 hours, i.e. no new runs are attemped after 7 hours. It turns out that our GP completed 30 runs in 25519.41 seconds, which is roughly 7.08 hours. On average the time needed for each run is 850.6 seconds. Also, f^* is obtained in a run which terminated after the 60th generation and this run takes 640 seconds. Thus our implementation of GP on dataset2 is efficient. It would be hard to evaluate the time need for each generations though, since the time used for generating a generation varies a lot depending on which generation it is.

Using h and g, the fitness and evaluation functions defined in the experiment section with rounding to 2nd digit after the decimal, we have

$$\begin{aligned} h(f^*) &= 7.79, \\ g(f^*) &= 0.91. \end{aligned}$$

Observe that $h(f^*)$ is larger than $g(f^*)$, which implicate that f^* doesn't encouter over fitting problem–it even approximate data points in test set better than in training set. Why is that? Well, remember that we did not choose the training set randomly. Among the 500 data points in the training set, 100 has very large output values and 100 has very small output values. As analyzed in the experiment section, these extreme points are very sensitive to the input and reflects properties of f better than normal points. This means two things: they are harder to approximate and may differ from the candidate more than general points, and they can help the smallest fitness to converge in faster speed. On the other hand, we also mensioned in Section Experiement that these extreme points tend to have more significant error in the dataset due to rounding. Thus $h(f^*)$ is larger than $g(f^*)$, which shows our selection of the training set is very successful.

Notice that h and g contain the regularization term to punish functions with high complexity. If we drop the regularization term, we see that the PMSE of f^* on the evaluation set (24631 data points) is 0.35. Remember these 24631 points are selected by ruling out points with x_3 or y less than 0.1. Since the dataset is rounded to 2 decimal place, this selection ensures the rounding error to be less than 5.2 percent. Remember PMSE is the MSE of the percentage difference of each data point. Thus, having PMSE of 0.35 means f^* on average differs from the real function by less than 0.6% on each data point in the evaluation set. This shows f^* is a good guess.

What about those 369 points that we ruled out? Among them, 24 points have either $x_3 = 0$ or $y = 0$. We will disregard these points, for different reasons. If $y = 0$, it causes two problems: first, when calculating PMSE, we need to divide the output y, so this may cause divide 0 error; second, since y is rounded to 0, the data itself already has 100% percent error, which makes PMSE meaningless. If $x_3 = 0$, there is arithmetically no good function that could approximate the output in the dataset. Our hand-guess function and the candidate function from GP both have the term $\div(x_3)^2$, which makes the output very sensitive to changes of x_3 when it's very small. For example, if we fix $x_1 = x_2 = 0$ and changes x_3 a little, then our candidate function gives

$$\begin{aligned} f^*(0, 0, 0.001) &= 1.23 \\ f^*(0, 0, 0.000000001) &= 1.23 \cdot 10^{12}. \end{aligned}$$

Clearly, 1.23 is very different from $1.23 \cdot 10^{12}$. However, since in dataset2 x_3 is rounded up to 2nd decimal place, it makes no distinction between these two input, i.e. input of $f(0, 0, 0.001)$ and $f(0, 0, 0.000000001)$ are both rounded to $f(0, 0, 0)$. Thus the data points with $x_3 = 0$ has no credibility at all and should never be used.

The PMSE among the rest $369 - 24 = 345$ points is 509.57, implying f^* differs from output value by 22.57%, on average. If we consider these 345 points together with the 24631 points in the test set, we have 24976 points with PMSE of 7.34, implying f^* differs from output value by 2.7%. We summarize these in Table 2, where *APE* is the percentage difference on average:

Restriction	Data Size	PMSE	APE
$y > 0.1, x_3 > 0.1$ (test set)	24631	0.35	0.6%
$0 < y \leq 0.1, 0 < x_3 \leq 0.1$	345	509.57	22.57%
$y > 0, x_3 > 0$	24976	7.34	2.7%
$y = 0$ or $x_3 = 0$	24	N/A	N/A
$y < 0$ or $x_3 < 0$	0	N/A	N/A

Figure 2: Evaluation of the best candidate f^* for dataset2.

In the experiment section, we described the difference between mutate functions for dataset1 and dataset2. We made this change because the mutate function in dataset1 (exchanging only constants to variables and variables to constants) behaves poorly on dataset2 due to its complexity. The mutate function defined for dataset2 (exchange a random node into a new subtree) gives more room for changes and behaves much better.

GP for Dataset3

The best candidate found by GP for this dataset, denoted by f^*, is

$$\begin{aligned} f^* =& ((((sin((((cos(x)) * (((3 - (sin(3))) \\ & + (sin((1 * 4)))) * (sin(2))))/(sin(((exp(5))/1)))))) \\ & - (sin((0 + ((4 + (((cos(x)) * (((2 - 4) + (sin((sin(x)))) \\ & * (sin(3))))/(sin((cos((sin((0 + 4)))))))) * 1))))) - x) \\ & * ((((cos(x)) * ((exp(((sin(x)) - x))) * (sin(x)))) \\ & /(sin(3))) + ((cos(x)) * ((4 - x) * (sin((3 * 4))))))) \end{aligned}$$

Our implemention of GP completed 300 independent evolutionary runs in 146501 seconds, which is roughly 40.69 hours. On average the time needed for each run is 488 seconds. Also, f^* is obtained in a run which terminated after the 161th generation and this run takes 11351.4 seconds. Notice that 11351.4 is much larger than 488, because we force most independent runs to terminate when the smallest fitness does not converge fast enough, as specified in the experiment section. This allows GP to excecute more independent runs in a reasonable amount of time and gives us a proper candidate. The MSE of this function is 21.16, showing that the value of f^* differs from the value of f by 4.6, approximately. Considering the range of $f(x)$ in dataset3,

$$-2917.159 \leq f(x) \leq 3729.416,$$

this error is relatively small and it shows f^* is a good guess.

Using h and g, the fitness and evaluation functions defined in the experiment section with rounding to 2nd digit after the decimal, we have

$$h(f^*) = 50.76,$$
$$g(f^*) = 85.76.$$

Unlike dataset2 where we do special selection for the training set, this time we have $g(f^*) > h(f^*)$, as expected.

Since this dataset has only one input, we can visualize f^* and the dataset with a general 2-dimensional graph, as shown in Figure 3. From this picture, it's clear that f^* is a good guess.

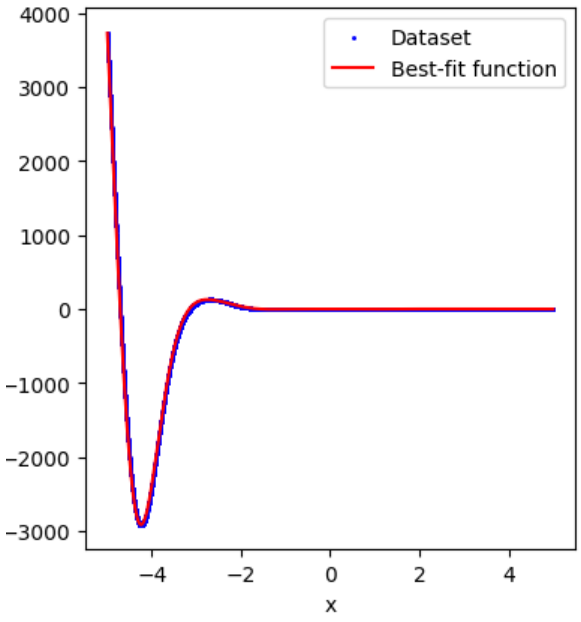

Figure 3: Plotting of best candidate f^* for dataset3.

5 Conclusions

In this paper, we explored the application of genetic algorithms in symbolic regression, with a focus on three distinct datasets. Our implementation of genetic algorithms found good candicate for each of these datasets, demonstrating the ability of genetic algorithms in handling intricate patterns of symbolic regression (multiple input, extended operators, etc.). The selected candidate functions exhibited low error rates and demonstrated their ability to capture complex patterns in the datasets.

Future work in this area could involve exploring additional datasets with different characteristics, refining the genetic algorithm parameters, and investigating the scalability of the approach to larger and more diverse datasets.

6 Contributions

We worked together to write the expression tree class. Richard implemented the genetic algorithm for data set 1, and Daniel implemented the genetic algorithm for both data set 1 and data set 2. Richard wrote the script for experiment and data collection while Daniel did the data engineering for data set 2 and 3. For the paper, Richard Wrote the introduction, conclusion,and background, while Daniel wrote the experiment and result. We both edited each other's work.

7 Acknowledgements

We appreciate the continuous guidance that Professor Raghuram Ramanujan provided.

References

Blickle, T., and Thiele, L. 1996. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation* 4(4):361–394.

Katoch, S.; Chauhan, S. S.; and Kumar, V. 2021. A review on genetic algorithm: past, present, and future. *Multimedia tools and applications* 80:8091–8126.

Koza, J. R. 1990. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*, volume 34. Stanford University, Department of Computer Science Stanford, CA.

La Cava, W. G.; Lee, P. C.; Ajmal, I.; Ding, X.; Solanki, P.; Cohen, J. B.; Moore, J. H.; and Herman, D. S. 2023. A flexible symbolic regression method for constructing interpretable clinical prediction models. *NPJ Digital Medicine* 6(1):107.

Pedace, R. 2013. *Econometrics for dummies*. John Wiley & Sons.