Genetic Programming-Based Symbolic Regression

Lindy Bustabad and **Caroline Sigl**
{libustabad,casigl}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

To perform symbolic regression on two data sets, we implement a genetic programming algorithm to select candidate expression tree models that best fit the observed data points. We generate individual trees and evolve them using well-studied techniques from genetic programming literature. The genetic programming implementation converged on the function $f_1(x) = x^2 - 6x + 14$ for dataset1 but was insufficiently robust to converge on a model for dataset2.

1 Introduction

Symbolic regression, also known as function identification, is a type of regression analysis. This regression technique searches the space of mathematical expressions in symbolic form in order to find a model that fits a finite sampling of independent variables and associated dependent variables with a prescribed precision. Candidate mathematical expressions are represented in symbolic form as expression trees with non-terminal nodes as mathematical operators and terminal nodes as constant values or variables. The expression trees are read in post-order traversal to represent each mathematical expression in the search space.

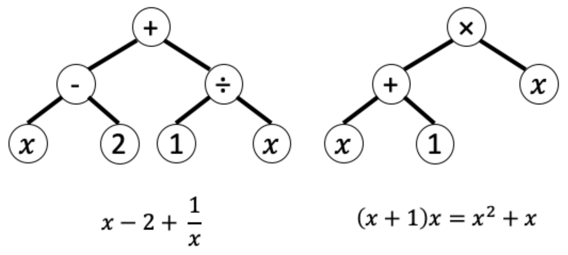

Figure 1: Parsed expression trees representing candidate functions.

Unlike traditional parametric regression, symbolic regression determines the functional form that best fits the data and computes the numerical coefficients after the appropriate type of function has been chosen. Symbolic regression techniques avoid influence by prior assumptions and instead infer the best model from the given data. One of the disadvantages to this approach is the infinite search space of models that perfectly fit a finite data set.

The algorithm for genetic programming-based symbolic regression was popularized by John Koza's research in 1992 (Koza 1992). Genetic programming is a computer search technique that uses genetic operators such as crossover, reproduction, and mutation to find a solution within a population of candidate programs. These operators are defined in Section 2. Genetic programming parallels natural selection, in which the initial population is generated at the start and only the most "fit" individuals survive and evolve between generations.

In this paper, we present an implementation of genetic programming-based symbolic regression. We perform symbolic regression on two data sets: *dataset1* and *dataset2*. Both of these data sets contain 25,000 data points corresponding to evaluations of a function. The dataset1 includes values for the variable x and their corresponding $f(x)$ values, and dataset2 includes values for the variables x_1, x_2, and x_3 and the corresponding measured values of $f(x_1, x_2, x_3)$. We implement methods developed by both Koza and Langdon et al. to combat problems of overfitting, bloat, and loss of diversity within our population of candidate expression trees throughout generational evolution (Koza 1992; Langdon et al. 1999).

This paper will document necessary background on the problem domain of genetic programming-based symbolic regression in Section 2. We will describe the implementation of genetic programming techniques and the experimental setup in Section 3. Finally, we will document our experimental results and draw conclusions on accuracy and deficiencies of the performed regression analysis in Sections 4 and 5.

2 Background

Genetic programming is a method of evolving computer programs to find a desired solution to a problem space. Populations of thousands of computer programs are genetically bred from an initial generation using the Darwinian principle of survival and reproduction of the fittest. Each individual in the population is a given a "fitness" score measured in terms of how well the individual performs in the problem environment. For symbolic regression, this fitness measure is based on how accurately an individual expression tree fits the given data. Subsequent generations are produced by sexual recombination (or crossover), genetic mutation, and duplication of individuals in the parent generation. Extensive research has been conducted on the many choices that can be made by the programmer when implementing genetic programming.

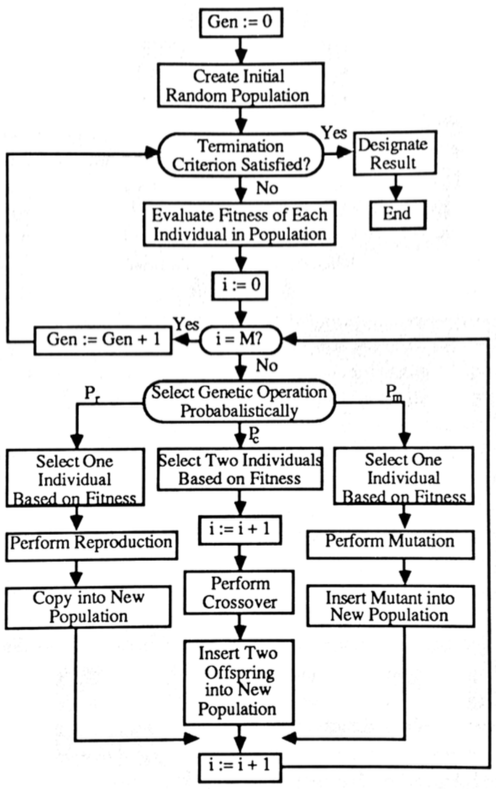
Gen := 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Designate Result
End
No
Evaluate Fitness of Each Individual in Population
i := 0
Yes
Gen := Gen + 1
i = M?
No
Select Genetic Operation Probabalistically
P_r
P_c
P_m
Select One Individual Based on Fitness
Select Two Individuals Based on Fitness
Select One Individual Based on Fitness
Perform Reproduction
i := i + 1
Perform Mutation
Copy into New Population
Perform Crossover
Insert Mutant into New Population
Insert Two Offspring into New Population
i := i + 1

Figure 2: Flowchart of conventional genetic algorithm (Koza 1992, Figure 3.1).

Initial Population

Researchers have identified several techniques for generating the initial "seed" population of individuals, one of which is the "ramped-half-and-half" method proposed by Koza. The ramped half-and-half method is a mixed generative method that incorporates two tree growth techniques: the *grow* method and the *full* method.

With the grow method, the population of expression trees are created one individual at a time up to a specified maximum depth, d_m. This is accomplished by making a random selection of each node label for points at depths less than d_m from either the set of operators or the set of terminals. Node labels at d_m are randomly selected from the set of terminals. This method ensures that trees are variably shaped.

In contrast, the full method creates trees by restricting the node labels for points at depths less than d_m to the set of operators and points at d_m to the set of terminals. This method ensures that all trees generated have a certain depth.

To increase variation in structure, the ramped half-and-half method divides the population into a total of $d_m - 1$ parts. Half of each part is produced by the grow method and the other half is produced using the full method. This generates an initial population with a proper range of randomly sized and randomly structured individuals (Koza 1992, p. 92-93). Additional techniques for grooming the initial population can be implemented, including "duplicate checking," or removing duplicate individuals from the initial population. Koza found that if duplicate checking is performed, the variety of the initial random population is one-hundred percent (Koza 1992, p. 93).

Fitness Function

Different versions of fitness functions have been tested in genetic programming problem domains. A commonly used metric for fitness is the *mean squared error* of a program that estimates some observed quantity. The mean squared error is a measure of the quality of an estimator model computed as

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

where n is the number of data points, y is the observed value of the variable being predicted, and $\hat{y}$ is the value of the variable predicted by the model. This value is always non-negative, and values closer to zero suggest a stronger correlation between the model and the dataset.

Overfitting Minimization and Regularization

A obstacle often encountered when performing symbolic regression is *overfitting* of candidate functions. As the genetic programming algorithm evolves functions across generations, the candidate expression trees have the potential to become excessively complex. In this case, it is possible that the learned hypothesis may fit the training set well but fail to generalize to new values, causing the model to have a low error on the data points but not the strongest *generalization error* (Artificial Intelligence - All in One 2016). To minimize overfitting, the data set can be partitioned into a "training set" and a "test set." When running genetic programming, individuals are evaluated on the training set. After generating the best models from each trial, the program then evaluates the models' performances on the test set to determine the most fit regression model for the data set. Because the genetic programming process does not make decisions based on the test data, the error on these data points is an unbiased estimate of the true performance of the regression model.

Another popular technique for minimizing overfitting in genetic programming is incorporating a penalty for overly complex hypotheses in the fitness function. This technique is called *regularization* as it encourages the evolution algorithm to attempt to fit the data set with simple models first before trying more complex models. The most successful algorithms have been shown to incorporate both mechanisms for managing overfitting.

Bloat Control

Another phenomena in genetic programming is the problem of *bloat*, defined as the increase in mean program size without a corresponding improvement in fitness. Several theories have been proposed in literature that attempt to explain why bloat occurs. One such theory is the Crossover-Bias Theory which claims the cause of bloat is that the distribution of program sizes during evolution is skewed in favor of larger programs by punishing smaller individuals (Trujillo et al. 2016). There are several proposed bloat control methods that attempt to limit the size distribution of individuals in the evolving population. The most common technique is restricting breeding to only produce children less than some maximal tree depth, popularized by Koza who set the depth restriction to 6 for the initial population and 17 for subsequent generations (Koza 1992, p. 114).

The *pseudo-hillclimbing*, or non-destructive crossover, method for reducing bloat was proposed by Langdon et al.. This technique rejects children if they are not superior to or different from their parents in fitness. If a child is rejected, the most fit parent is copied to the next generation. This method replicates large numbers of parents into later generations, slowing the growth in average model size. Langdon et al. found that the use of pseudo-hillclimbing significantly lowers the amount of code growth without significant change in fitness (Langdon et al. 1999).

Control Parameters

There are additional decisions to be made in the design of the evolutionary operations of the genetic programming algorithm. The suggestions for optimal population size, number of generations, and crossover, mutation, and reproduction rates for the evolutionary process vary throughout literature. Koza fixes the following parameters in his pioneering work on genetic programming:

POPULATION SIZE	500
GENERATIONS	51
CROSSOVER RATE	0.90
MUTATION RATE	0
REPRODUCTION RATE	0.10

Table 1: Genetic programming parameters in Koza's research (Koza 1992, p. 114)

The crossover method combines genetic material from two parents to form two new individuals for the new population. In the problem domain of symbolic regression, crossover switches a randomly selected subtree of one parent expression with the subtree of the second parent expression to produce two child expressions.

The mutation operation alters a randomly selected node of an individual expression tree. This operation can change the solution entirely from the previous solution. Thus, it is possible the genetic programming algorithm will have access to a more fit solution after mutating part of the population. In this way, mutation has the potential to restore lost diversity in a given population.

Additionally, reproduction is an asexual operation in which a selected individual copies itself into the new population. This represents an individual surviving into the next generation.

The selection of an individual to undergo reproduction, crossover, and mutation is the responsibility of the *selection function*. Koza uses a number of selection functions, including fitness proportionate selection, greedy over-selection, and tournament selection in particular. The implementation in this paper uses only the fitness proportionate selection, or roulette wheel selection. With this selection function, the chosen fitness function assigns a fitness to all candidate functions. This fitness is associated with a probability of selection computed as

$$p_i = \frac{t_i}{\sum_{j=1}^{n} t_j}$$

where n is the number of individuals in the population and t_i represents the fitness of individual i. This can be imagined similar to a roulette casino wheel. A proportion of the wheel is assigned to each candidate function based on their probability associated with their fitness. A uniform random number is chosen between 0 and the maximum of the probabilities and this corresponds to the roulette ball falling in the bin of an individual with a probability proportional to its width (Walker 2001).

3 Experiments

In our research, we experimented with parameters and evolution controls in order for the genetic programming algorithm to consistently converge on a model with each trial run. The program generated ten models from ten independent runs of the evolutionary algorithm and the algorithm selected the best overall performer among these models. We ran the program ten times for each data set, effectively performing one-hundred independent runs of the algorithm.

Initial Population

We implemented Koza's ramped half-and-half method to generate the initial population of expression trees prior to evolution. To preserve diversity in the initial population and encourage effective evolution, we removed individuals that had fitness values of either 0 or undefined from our initial population. Additionally, we implemented duplicate checking and continued generating individuals until reaching the specified initial population size.

Fitness Function

Once the initial random population was created, we assessed individuals for their fitness by calculating the mean squared error of each candidate function. We included a penalty for deeper trees by scaling the mean squared error by a factor of 1.05^d where d is the depth of the tree, giving higher values for deeper trees. This adds an evolutionary pressure pushing the algorithm towards less complex solutions and controlling for code bloat. We found a factor of 1.05^d to be the most effective in supporting convergence after incremental testing of various factors ranging from as large as d to as small as 0.1^d. We assigned the reciprocal of the scaled mean squared error as the fitness for each function, giving higher values for desirably smaller trees. Our fitness function was measured as

$$\frac{n}{1.05^d \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

where n is the number of data points, y is the observed value of the variable begin predicted, and $\hat{y}$ is the value of the variable predicted by the model.

Overfitting Minimization and Regularization

We partitioned the first two-thirds (16,666 data points) of the given dataset into the training set and the final one-third (8,334 data points) into the test set. Our genetic programming implementation evaluated the fitness of individuals based on the training set and selected the best overall performer by evaluating based on the test set. We introduced the penalty for overly complex hypotheses to encourage the algorithm to fit the data with simple hypotheses before complex ones.

Bloat Control

We introduced Koza's depth restriction technique to prevent code bloat in our genetic programming algorithm. We attempted to replicate Koza's restrictions of depth 6 for the initial population and depth 17 for subsequent generations; however, this slowed the evolutionary search process and consumed excessive memory. We incrementally decreased these restrictions until reaching a manageable program runtime. We established a maximum depth of 4 for the initial population. For subsequent generations, we did not restrict the depth of candidate functions, but rather assigned fitnesses of 0 to functions of depth greater than 10, preventing these trees from being selected in future generations. Additionally, we implemented pseudo-hillclimbing in the crossover operation of each generation.

Control Parameters

We began our experiments by utilizing Koza's established parameters for genetic programming. Through incremental testing, our research found a correlation between a 0.1 percent mutation rate and a higher probability of convergence on a consistent regression model.

POPULATION SIZE	500
GENERATIONS	51
CROSSOVER RATE	0.90
MUTATION RATE	0.001
REPRODUCTION RATE	0.10

Table 2: Genetic programming parameters implemented.

Selection Function

We implemented fitness proportionate selection to choose individuals from the population for crossover, mutation, and reproduction.

4 Results

Running our program with the parameters and methods described in the previous section, we were able to converge on the following equation for dataset1:

$$f_1(x) = x^2 - 6x + 14.$$

We obtained this result by simplifying the equivalent equations produced by ten separate runs of our program on dataset1 shown below:

Equivalent Function	Fitness	Gen
$x - 11 + (x-4)(x-3) + 13$	1327.046	1
$(-5+x)(-2+x) + 4 + x$	1327.046	2
$x + (x-2)(x-4) - x + 6$	1327.046	4
$x^2 + 4 - 5 - x - 5x + 15$	1327.046	4
$(-4+x)(x-2) + 6$	1263.854	6
$(\frac{5}{3} + 3 - x)(2-x) + \frac{2x}{3} + \frac{14}{3}$	1263.854	9
$3x + 17 - x(8-x) - x - 3$	1203.670	9
$-((x-4)(1-x) - 11) - x - 1$	1146.754	13
$\frac{-1}{4} + x^2 + 3 + (13 - \frac{7}{4} - 6x)$	1263.854	19
$(-5+x)(x-2) + 4 + x$	1393.399	23

Table 3: Functions converged on by the genetic programming algorithm for dataset1.

The algorithm found the candidate function $f_1(x)$ as early as the first generation and as late as the twenty-third generation over the course of ten trials, or one-hundred runs of the genetic programming algorithm. Functions with the same fitness have equal depths. Figure 4 shows the increase in fitness values of candidate functions for a trial that found $f_1(x)$ in generation four. The highest fitness value reported when the algorithm converged on $f_1(x)$ was 1327.046. The algorithm successfully performs natural selection to evolve candidate functions with higher fitness values and avoid reproducing those with lower fitness values. Figure 5 shows the decrease in mean squared error of candidate functions for the same trial. The mean squared error for individuals in the population significantly decreased after the first generation, suggesting the robustness and efficiency of the genetic algorithm's evolutionary operators in producing subsequent generations for dataset1. The mean squared error fell to 0.0006 for this trial, suggesting $f_1(x)$ is a strong fit for dataset1.

Our algorithm was not robust enough to converge on an equation for dataset2. The following function is indicative of the most fit candidate functions selected by the genetic algorithm:

$$f_2(x) = \frac{123x_1x_2x_3 + 41x_3}{6x_1x_2 + 3x_2x_3}.$$

The mean squared errors of candidate functions remained extremely high, suggesting these models are poor fits for dataset2. Figure 6 shows the increase in fitness values of candidate functions for five trials of the algorithm on dataset2. Overall, the fitness values are increasing over the course of fifty generations. Figure 7 shows the decrease in

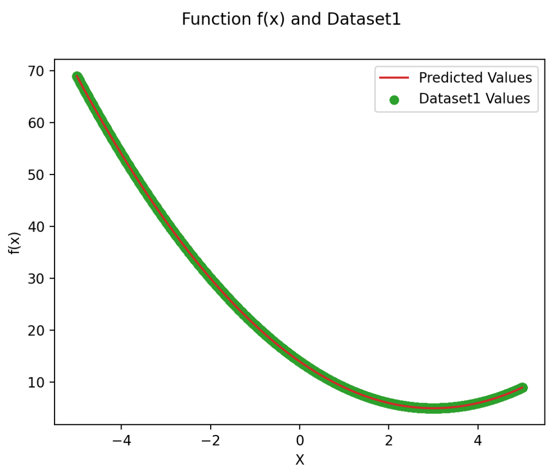

Figure 3: Plot of dataset1 values and $f(x)$ regression model.

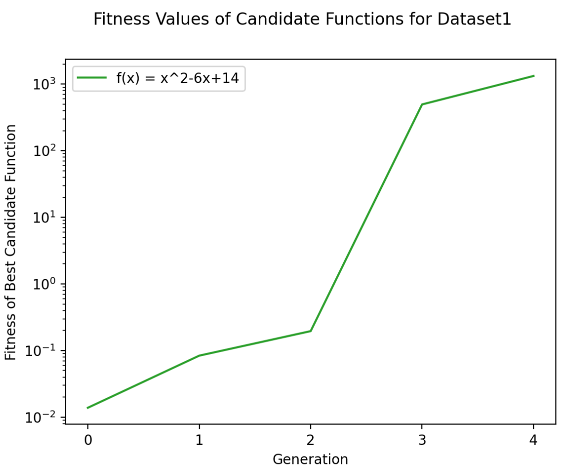

Figure 4: Fitness values for the best candidate function in each generation up to convergence for dataset1.

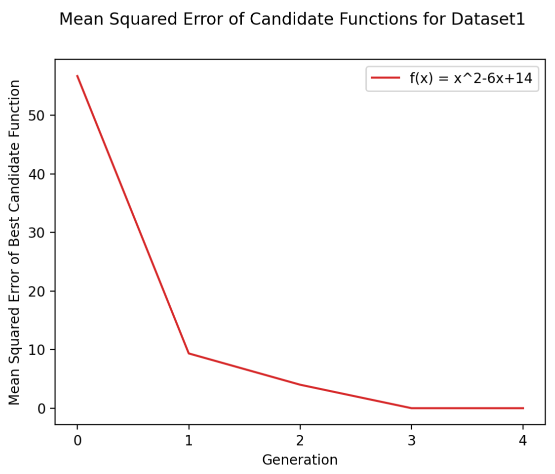

Figure 5: Mean squared error values for the best candidate function in each generation up to convergence for dataset1.

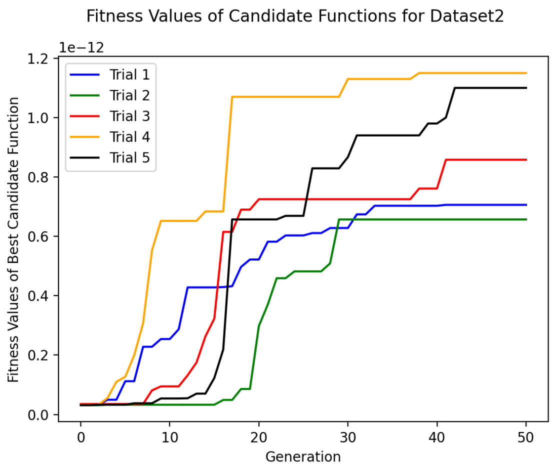

Figure 6: Fitness values for the best candidate functions in each generation for five trials.

mean squared error values of candidate functions for the same five trials on dataset2. Similarly, the mean squared error values are decreasing over the course of fifty generations. These trends suggests that the algorithm is attempting to introduce natural selection in each generation and evolve models with higher fitnesses and lower mean squared errors in order to converge on the most fit model.

The average $f(x_1,x_2,x_3)$ value for dataset2 is $f(x_1,x_2,x_3) = 6.834\times10^6$. We used the line $f(x_1,x_2,x_3) = 6.834\times10^6$ as a model for dataset2. The mean squared error achieved by this line is 1.034×10^{18}. Additionally, the median $f(x_1,x_2,x_3)$ value for dataset2 is $f(x_1,x_2,x_3) = 52.345$. Similarly, the mean squared error achieved by this line is 1.038×10^{18}. Thus, the lowest mean squared error achieved by a straight line is 1.034×10^{18} for dataset2. The lowest mean squared error for a candidate function found over the course of our five trials was 6.49×10^{11}. Thus, while a mean squared error of 6.49×10^{11} shows that our model has clearly not converged to an optimal solution, we outperform a straight line and therefore the genetic algorithm is beginning to train on the data in order to find a convergent function for the data set. However, it is unclear how much lower the mean squared error of a strong model for this data set might be because even a “perfect” fit function may not have a mean squared error of 0 due to noise in the data set.

It can be observed that the fitness values sharply increase and the mean squared error values sharply decrease over these trials between generations 17 and 20. After generation 20, the values change much more slowly and plateau after reaching their optimal values. There are several reasons as to why this might occur. The first is the possible loss of diversity in the population. It is likely that the population of candidate functions becomes homogeneous after generation 20, meaning all of the functions evolve to rearrangements

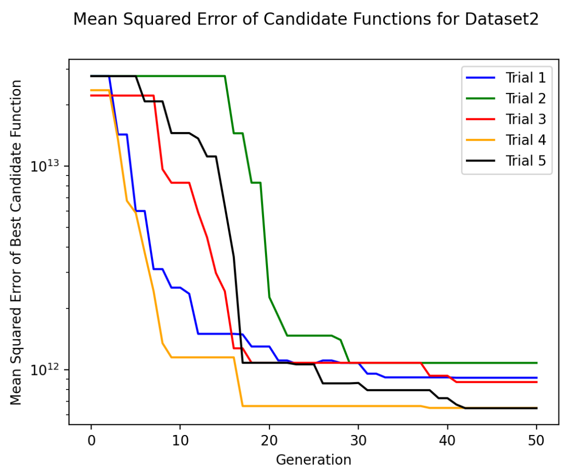

Figure 7: Mean squared error values for the best candidate functions in each generation for five trials.

of each other. This could mean the crossover and mutation functions are not robust enough to continue introducing diversity into the population with each new generation. We attempted to lower the reproduction rate of the algorithm to decrease the amount of direct copies of individuals that are added to each new generation, but this did not impact our algorithm's performance.

Another possible reason that the fitness values did not improve past generation 20 is that we did not allow for sufficiently complex functions in each generation for dataset2. Because we assigned fitnesses of 0 to trees with depth greater than 10, these trees were never selected. For dataset1, the convergent function was able to be modeled with a simple function. It is likely that the best-fitting model for dataset2 is more complex and our algorithm was unable to discover this model due to our depth-limiting fitness penalty.

5 Conclusions

Our research implements genetic programming-based symbolic regression on two datasets. We introduced genetic programming techniques proposed by Koza and Langdon et al. to support the genetic algorithm in converging on a best-fit model for the given dataset. Our implementation successfully converged on a model for dataset1, but was unsuccessful in converging on a model for dataset2. The algorithm's failure was likely caused by loss of diversity in later generations. In order to allow for more complex trees while still attempting to find less complex trees first, future research should implement iterative deepening depth-first search to flatten fitness values at iterative depths rather than an established depth. The control parameters and techniques used can possibly be improved by increasing the population size, generation limit, and evolutionary operation rates with each trial. Additionally, the implementation could be adapted to add a subset of random individuals to each generation to reintroduce diversity into each generation.

6 Contributions

C.S. wrote the tree classes to read in data and initialize the population of individual functions for datasets 1 and 2. Both C.S. and L.B. designed the fitness and selection functions. Both C.S. and L.B. wrote the genetic programming algorithm to produce each generation. L.B. wrote the crossover, mutation, and reproduction methods. Both C.S. and L.B. ran trials for datasets 1 and 2 to collect data. L.B. wrote the background, experiments, and results sections. C.S. wrote the introduction, conclusions, contributions, and acknowledgements sections. C.S. prepared tables 1 and 2. L.B. prepared figures 1, 2, 3, 4, 5, 6, and 7. L.B. prepared table 3. Both authors proof-read the entire document.

7 Acknowledgements

We would like to thank Dr. Raghu Ramanujan for his assistance with code design, analysis of experimental results, and advice on reporting research findings.

References

Artificial Intelligence - All in One. 2016. Lecture 7.1 — regularization — the problem of overfitting — [machine learning — andrew ng]. [YouTube video]. Accessed Oct. 3, 2020.

Koza, J. R. 1992. Genetic programming : on the programming of computers by means of natural selection.

Langdon, W.; Soule, T.; Poli, R.; and Foster, J. 1999. *The evolution of size and shape*, volume 3. 162—191.

Trujillo, L.; Muñoz, L.; Galván-López, E.; and Silva, S. 2016. neat genetic programming: Controlling bloat naturally. *Information Sciences* 333:21 – 43.

Walker, M. 2001. Introduction to genetic programming.