Symbolic Regression using Genetic Programming

Joshua Wallace and **Wilbert Garcia**
{jowallace,wigarcia}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

In this paper, we implemented symbolic regression using genetic programming to compare the efficiency of different parameter configurations on fitting data from two different data sets. We successfully implemented our genetic algorithm on the first dataset, but not on the second dataset. Through the analysis of different algorithm configurations, we determined that a parameter set with a smaller population size, a wider range of constants, and a limited elite size resulted in the best performance.

1 Introduction

Genetic algorithms implement a stochastic hill climbing approach. The algorithm creates a population of viable solutions to the problem at hand, in this case, symbolic regression, to determine a function that best fits a dataset. Each viable solution is referred to as a chromosome in a population and all chromosomes are competing for reproductive success. The success is determined through a fitness function that measures how well the function fits one of our datasets. This creates a new population of more fit individuals that undergo mutation and crossovers to create a new population of ideally more fit chromosomes that continue to compete.

To select parents for following generations, our genetic program uses Elitism to determine the fittest chromosomes from a population that will continue to the next generation. A consideration in determining which individuals will move on is the potential for overfitting where the selected function may fit only the dataset well but does not generalize well to other data. In order to minimize overfitting, we have used a train/test split to account and test for overfitting and have implemented a penalty on the total number of nodes in a tree when calculating fitness. (Bronshtein 2017)

This paper investigates different sets of parameters that optimize the performance of the genetic program in creating a function that fits the data well. We ran experiments that allowed us to determine the set of parameters that optimized the effectiveness of the genetic program.

We ran these experiments on two different datasets. Both datasets have 25,000 datapoints. The first dataset has an x variable value and a corresponding value of a function. The second dataset has 3 different x variable values and a corresponding value of a function.

2 Background

grow() The grow function is called on a node containing an operator as a value. This function randomly decides the value for both the left and right child of the node on which it is called from either the operator set or terminal set. The grow function is then called on each of the newly generated children nodes which have a value in the operator set. This recursive expansion will continue until all leaf nodes of the tree have values from the terminal set.

findFitness() Each tree has a fitness property which is determined by the findFitness function. The fitness of each tree is equal to the root mean squared error for a given tree over the x and f(x) values of the given dataset, with an additional penalty for each node in the tree. This penalty encourages more concise solutions. The lower the fitness value, the more fit a tree is.

Trees can be altered in two ways, mutation or crossover.

1. mutate() The first method of altering trees is the mutate function. This function selects a random node in the tree and swaps its value for a different value in the same set of values to which the original belonged. If the randomly selected node was originally an operator, a new operator will randomly be swapped in. If the node was originally a terminal it will be exchanged for a different terminal value.

2. crossover() The crossover function exchanges pieces of two trees to create two new trees. The crossover function selects a random node in both of the trees being crossed and then exchanges the two nodes.

eliteSelection() This function runs through all of the trees in a given population and saves the most fit trees it finds that generation.(Du et al. 2018) These trees are then used as the parents for creating a new generation in the newPopulation function.(Chudasama and Panchel 2011) newPopulation() This function utilizes the list of parents generated in eliteSelection() to create a new generation of trees. A random float is generated between 0 and 1. Based on where that random float falls between the parameters crossover percent and mutate percent,a new tree is generated using crossover, through mutation, or if the number falls outside the range for both mutation and crossover, newPopulation() simply grows a random new tree. This encourages randomized diversity in the population. Additionally, if the population is determined to be stale, newPopulation() keeps the parents, but will populate the rest of the subsequent generation through the growth of randomized trees.

createPopulation() Creates an initial seed population of the size dictated by the population size parameter. Generates a one node tree with a random operator as the value, then calling grow on that node to make a full tree. This is done until there exist population size number of trees.

Definitions

- Terminal Set: The terminal set is defined as T = $\{x, \hat{A}\}$ where x represents the given variable and $\hat{A}$ denotes constant numerical terminals in the range constant A to constant B.
- Constant A:Lower bound of possible numbers in terminal set.
- Constant B: Upper bound of possible numbers in the terminal set.
- Operator Set: The operator set consists of basic arithmetic functions and is defined as O = {*, /, +, - }.
- Depth: Parameter indicating the maximum depth to which trees are permitted to grow. If the tree reaches this depth, all new children are taken from the terminal set so that the tree stops growing.
- Elite Size: The number of trees carried over between generations.
- RMSE: Root mean squared error - Determine the deviation between the desired f(x) and the result from entering x into the equation represented by a given tree, and then square the result. Sum up for all x and f(x) pairs in the given dataset, find the average, and then take the square root of that average.
- Population Size: The number of trees per generation.
- Mutate Percent: Float value which determines how frequently a mutation occurs in creating a new population.
- Crossover Percent: Float value which determines how frequently a crossover occurs in creating a new population.
- Stale: Boolean value which indicates if the algorithm has not found a new best tree in five consecutive generations.

3 Experiments

Our goals for this experiment were to implement symbolic regression through genetic programming in an effort to determine the optimal parameters for maximizing efficiency of the genetic programming solution. In our genetic programming solution, there are numerous parameters which impact efficacy. We measured efficiency by analyzing our genetic algorithm's ability to produce a function that minimized mean squared error (MSE) on two different datasets.

We accomplished this through representation of equations as binary trees, performing mutations and crossovers between trees to work toward an equation that best fits the provided data points. Through analysis of the time taken and the changes in fitness, we were able to evaluate the given sets of data as well as determine the optimal set of parameters.

To collect data, we varied our parameters and tested different variations of our parameters. Our results were collected using the hardware capacity of a standard Mac laptop. We varied the size of the population, the range of constants that were included in the trees of the genetic algorithm, the maximum depth of the trees, the number of trees that are selected through elite parent selection, the percentage of trees in a population that undergo crossover and the percentage of trees that undergo mutation.

To test these parameter variations, we ran each of them 20 times and collected data on the MSE for each of these runs to determine an average MSE for each parameter setting. We used the average MSE to determine which parameters were the best in producing a function that fit the data.

4 Results

Our results indicate that there were certain parameter sets that outperformed other parameter sets when tested on dataset 1. We were able to determine that parameter set 2, with a population size of 100 trees per generation, an elite size of 10, and integers between -10 and 10 outperformed the rest of the parameter sets we tested. As seen in figures 1, 2 and 3, our algorithm performed better with a wider range of integers in the terminal set and with a population size to elite size ratio of 10:1. Parameter set 3 performed the worst in terms of average MSE. Parameter set 3 featured a population size of 500 and integers between -5 and 5. Varying the elite size from 10 percent of the population size to 20 percent of the population size resulted in the difference in performance between parameter set 3 and parameter set 2. There was a clear difference in average fitness between both of these parameter settings likely because an increase in the elite size results in higher diversity of functions that are being considered for the new population from generation to generation. For our algorithm, performance was improved by a decreased population size, an increase in the range of constants and increase in the percentage of the population considered in elitism parent selection.

Our results were limited by the run-time of our algorithm. The sixth parameter setting was testing the number of trees in the population. After 24 hours, it had computed three of the twenty generations we set as the baseline for comparisons, making the results for the parameter configuration inconclusive. Using our algorithm, 1000 trees per generation far exceeded the capabilities of our hardware.

Our use of a train/test set minimized overfitting in our algorithm.(Bronshtein 2017) Based on examination of the fitness of the trees being created, the train/test set was successful and our results did not show indications of overfitting the data.

Loss of diversity was certainly of concern with our algorithm. As is seen in figures 2 and 3, when we increased our population size significantly but did not increase the elite size, our algorithm performed worse on average than the rest of the parameter sets, likely indicating a loss of diversity. However, when we decreased the population size to 50 while keeping the elite size at 10, the algorithm did not perform significantly better. In fact, the sets with a proportionately greater elite size (sets 4 and 5) did not perform better on average than set 2 with a population size of 100 and an elite size of 10. While set 2 had a longer run time than sets 4 and 5, due to the extra required calls to findFitness() every generation, the 10:1 ratio seen in parameter set 2 performed on average the best out of all the sets we tested. Therefore, we believe our elite size parameter was properly set to minimize loss of diversity. Additionally, our stale parameter aims to prevent the algorithm from getting stuck building off of a bad set of parents.

Our results were inconclusive for the second dataset. We were unable to effectively produce functions that fit the data of the second dataset. We calculated standard deviation and variance for the output values of both datasets in an effort to determine how dispersed the data was in order to better understand its randomness. For the first dataset, there was a standard deviation of 18.882042 and a variance of 356.54577 . For the second dataset, there was a standard deviation of $2.886415e+06$ and a variance of $8.331726e+12$. The standard deviation and variance for the second dataset were very high and indicate significant dispersion of data making it more difficult to find an equation that effectively fits the dataset. The standard deviation and variance of the first dataset were high but we were still able to produce and equation that effectively fit the data.

Best-Performing Models

- Dataset 1: f(x) = (x^2 + 8) + (6 - (6x))
- Dataset 2: Not able to get close enough fit to have a prediction.

Figure 1: Parameter Test Sets

Set	Population Size	Generations	Const. A	Const. B	Depth	Mutate Pct.	Crossover Pct.	Elite
1	100	20	-5	5	20	0.2	0.8	10
2	100	20	-10	10	20	0.2	0.8	10
3	500	20	-5	5	20	0.2	0.8	10
4	50	20	-5	5	20	0.2	0.8	10
5	50	20	-5	5	20	0.4	0.8	5
6	1000	20	-10	10	20	0.2	0.8	10

Figure 2: Average Fitness on Dataset 1

Set	Average Fitness
1	10.37
2	9.57
3	12.26
4	11.72
5	10.46
6	Did not complete

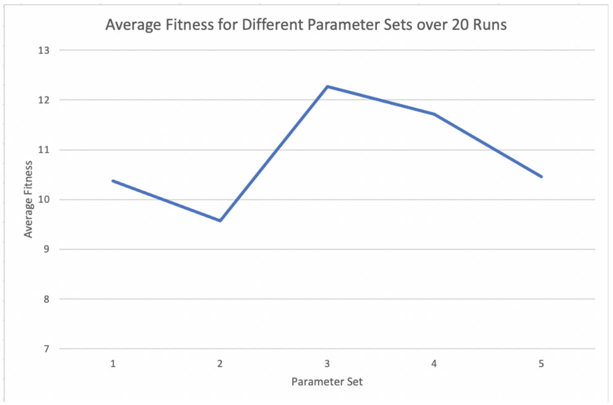

Figure 3: Average Fitness on Dataset 1

5 Conclusions

We set out to represent symbolic regression through the implementation of a genetic algorithm in order to produce a function that best fits a given dataset, optimize the performance of our genetic algorithm through parameters, and replicate and build on previous work in the field. We were able to successfully implement our genetic algorithm on the first dataset to solve for an equation which fit the dataset. However, for the second dataset, our algorithm failed to produce an equation that effectively fit the data, which we attribute to the significantly greater spread of the data in dataset 2. We also were successfully able to test the parameters of our genetic program in order to refine and determine more optimal settings for our algorithm.

A potential concern is that the fitness function and how it is called throughout our program results has an extremely high time-cost. To continue our work we could improve the efficiency of fitness calculations and our algorithm as a whole, and continue to optimize the parameters.

6 Contributions

- Wilbert Garcia: Random node, mutate, new population, and clone functions. Determined test method and wrote introduction and abstract.
- Joshua Wallace: Crossover, fitness, and evaluate functions. Wrote background, created citations, and made graphs.
- Collaborative: Elite selection, grow, new population, and genetic algorithm functions. Wrote experiments, results, and conclusions.

7 Acknowledgements

Thank you Dr. Ramanujan for the help and guidance.

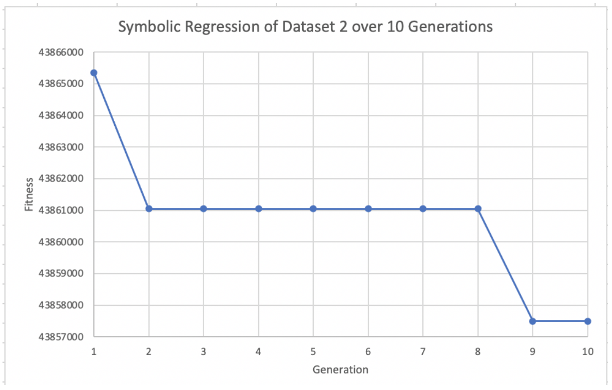

Figure 4: Fitness Over 10 Generations on Dataset 2

References

Bronshtein, A. 2017. Train/test split and cross validation in python. `https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6`.

Chudasama, Chatan, S. S., and Panchel, M. 2011. Comparison of parents selection methods of genertic algorithm for tsp. *International Journal of Computer Applications*. From International Conference on Computer Communication and Networks 2011.

Du, H.; Wang, Z.; Zhan, W.; and Guo, J. 2018. Elitism and distance strategy for selection of evolutionary algorithms. *IEEE Access* 6:44531–44541.