Symbolic Regression

Nakul Upadhya, Tyler Messerly

Date: 12/1/2021

Contents

1 What is Symbolic Regression

Symbolic regression is the process of identifying mathematical expressions that fit a given dataset. This method of regression differs from other forms such as Linear Regression, Ridge Regression, LASSO, etc. in that we do not assume the shape of the data. Instead, we theorize a set of operations (ex. log, exponent, addition, division, etc.) and features that may contribute to a response variable. Then various methods can be used to organize these questions into a potential governing equation

In this project, we will use genetic programming to construct our end equation. Each organism's genetic code will be represented by a computational tree that consists of the defined operations. The population will then undergo evolution.

2 GP Design and Experimentation

In order to perform symbolic regression on our given data sets, a genetic program was created using the `deap` package in python. Each data set was randomly into a training set (80% of the data) and a testing set (20% of the data). The evolutionary process was done using the training set, and the quality of our solution after the genetic program finished was evaluated using the testing set. The fitness function for a given solution was simply returning the squared error, and the algorithm tries to minimize the squared error.

Our GP had a population of 300 and ran for 2000 generations. The evolutionary strategy chosen was $\mu + \lambda$. After experimentation, this evolutionary strategy had a fitness that was magnitudes higher than the fitness of the simple algorithm. This strategy however had a much longer runtime than the simple algorithm.

The operation set for our genetic program are: addition, subtraction, multiplication, protected division, negation, cosine, sine, generate integer ephemeral constant, generate a factor of pi ($c\pi$) as an ephemeral constant, generate a factor of Euler's constant (ce) as an ephemeral constant. While we attempted to add power operators and log / ln to our operator set, both of these operations resulted in the algorithm crashing (either due to extremely large numbers or invalid operations like ln(0)), therefore they were excluded.

A key piece of the design parameters was the range of the constant operators. When first developing the genetic program, the ephemeral constants $c \in \{-1, 0, 1\}$, $c\pi \in \{-\pi, 0, \pi\}$, $ce \in \{-e, 0, e\}$. This initial range resulted in very bloated trees and extremely complicated equations. One possible reason for this was that the GP might have gotten the right *shape* of the data, but discarded that solution since the coefficients for the various terms were very incorrect. To rectify this, we increased the range of c from -1 to 1 to -50 to 50 ($c \in \{-50, 0, 50\}$.

With this increase in range, a change in mutation rate needed to occur. In the baseline GP, our initial mutation rate was 0.1 and our crossover was 0.9. When running our algorithm with these parameters and the higher range of constants, we often noticed that the algorithm got stuck in local optima for long periods of time and that the fitness did not improve for hundreds of generations. We attributed this to the fact that there is a "larger" space to explore now with the constants. To effectively explore the values of the coefficients on our various terms, the mutation rate needed to become higher. After experimentation, a good mutation rate was 0.3. This came with a crossover rate of 0.7 since the $\mu + \lambda$ evolutionary strategy requires that the mutation and crossover rates sum up to one. The mutation method was a uniform mutation, where a node on the tree is replaced with a randomly generated sub-tree. The crossover method was a single-point crossover.

A large piece of the symbolic regression design is controlling the height of our syntax tree. The two methods used to control this are a maximum tree height constraint, as well as a double selection method. During our experimentation, the maximum tree height was varied to find an ideal point. With an extremely low tree height ($h \leq 4$), the GP often got stuck in local optimal till the end of the program. We theorize that this is because each mutation (adding a sub-tree) can barely change the tree since the height of the sub-tree will be very small. On the other hand, having a large maximum tree height ($h \geq 10$) had very low error, but also outputted extremely complicated equations that performed poorly on our testing set. We found a good balance at a maximum height of 7. This maximum height worked well for both of the data sets. The other method for bloat control was our selection method. Compared to using a roulette or a tournament based on only fitness, the Double Tournament selection method where solutions were compared on not only fitness greatly reduced the bloat of the solutions. We noticed that using a fitness-only method caused most of the solutions to have a size very close to the maximum possible size. In comparison, the average size of the population when using the double selection method was very low in comparison. However, in order to ensure that the most accurate solution is always in the next generation, we implemented elitism.

To test our algorithm, we used the configuration described above and ran the genetic program 30 times on each training data set. We then took the best individual from each run for each data set, found the best performer

on the training data, and evaluated each individual on our test data. All of our code can be found at `https://github.com/upadhyan/Symbolic-Regression`.

3 Data Set 1

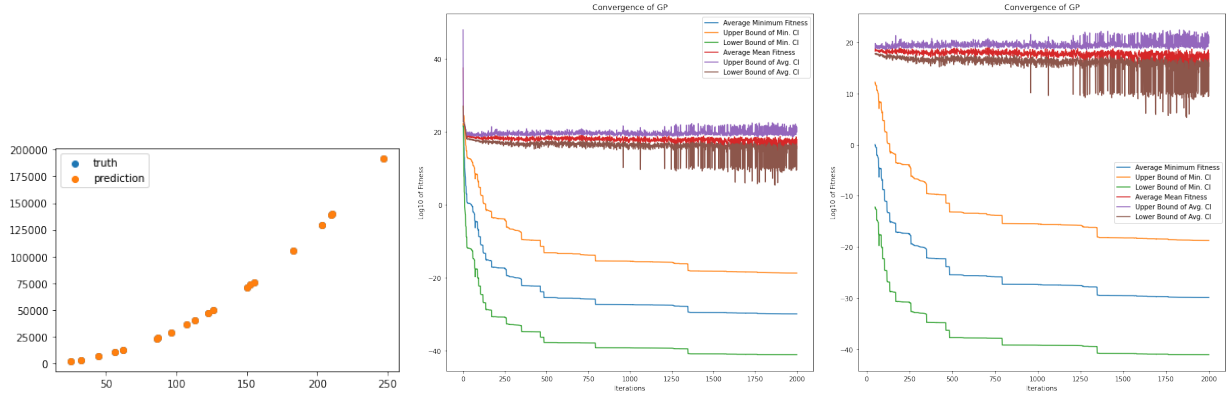
When testing our algorithm on the first data set, the "best" equation found was:

$$y = x_1(-3.85 \times 10^{-15} x_1 \sin(17.85 x_1) + 3.14159265 x_1 - 1.158 \times 10^{-15} \frac{50 - x_1}{\sin(x_1)} - 1.033 \times 10^{-13})$$

The squared error of this equation on our training set was 5.744×10^{-20} and the squared error on our training set 8.369×10^{-20}. Looking at the above equation, we can simplify it a bit to try and find the actual governing equation of the data. Many of the terms here are extremely small, therefore we can just remove them as that might be a result of noise. This gives us a *very familiar* equation: πx_1^2. It looks like x_1 is the radius of the circle, and y is the area of the circle.

When looking at the prediction and the truth plotted together, we can see that the prediction is extremely close to the true values of the data, so much that we can't see the true points.

Along with the end solution, we wanted to examine the convergence behavior of our genetic program. To do this, we plotted the mean of the average fitness per iteration for all runs and the mean of the minimum fitness per iteration for all the runs. For both of these metrics, we also plotted the upper and lower bounds of their confidence intervals.



(a) DS 1 Prediction and Truth (b) All Iterations (c) Iterations ≥ 50

Figure 1: Convergence Graphs for Data set 1

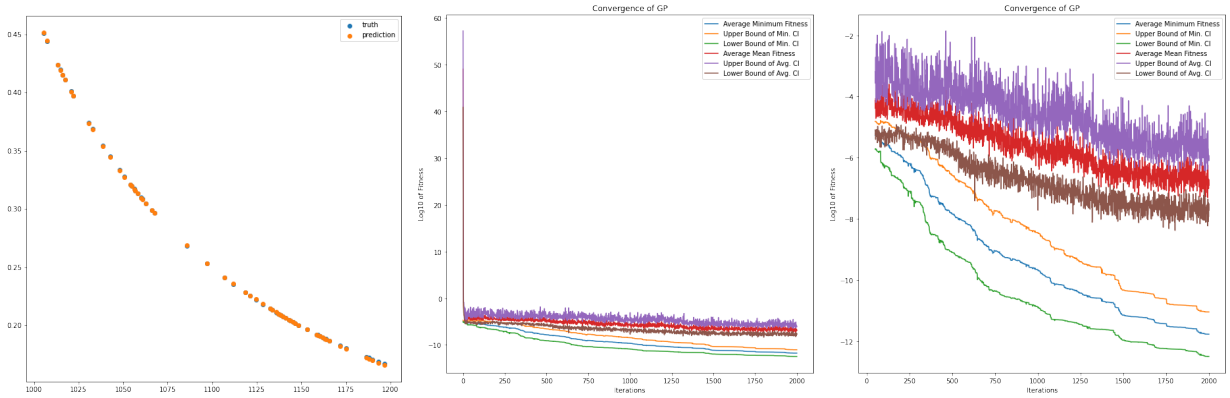
Looking at these figures, we can see that while the average fitness of a run remains relatively constant (give and take some noise), the minimum fitness experiences a gradual decline. Additionally, the frequency of the discoveries (marked by sudden drops in the minimum fitness line) is much higher in the first 500 iterations compared to the next 1500 iterations.

4 Data Set 2

The equation outputted by the genetic program for the second data set was:

$$\frac{1.947x_1 - .106x_3 + 3}{-x_1 + x_2 + 1.040x_3 + 6.283}$$

Unlike the previous equation, there is no real discernible equation here. Adjusting the height of the tree did not help make this equation simpler at all.



(a) Prediction Fit (b) All Iterations (c) Iterations ≥ 50

Figure 2: Convergence Graphs for Data set 2

The squared error of the above equation on our training set is 1.864×10^{-7} and the squared error on our testing set is 1.311×10^{-7}. Unlike data set 1, the testing error for the equation produced by this data set is lower than the error on our training set. This could indicate that our genetic program needs more generations to fully converge. This is supported by figure 2c since it can be seen that the slope of the minimum convergence line has only just started to decrease (ignoring the massive decrease in the first 50 iterations).