Natural Computing Project Assignment 1 - Genetic Programming for Symbolic Regression

Breno Campos Ferreira Guimarães

September 30, 2018

Abstract

In this report, we describe the implementation of a Genetic Programming algorithm towards solving the problem of Symbolic Regression. In these problems, the goal is to find a mathematical function that best fits a set of observations, given no previous knowledge of the function's nature (degree, continuity, etc). We first describe the manner in which the problem was modeled as an instance of Genetic Programming, and go over a few important implementation decisions. Then, we perform a thorough analysis of the execution parameters, which describes how the ideal values were chosen empirically.

1 Introduction

In the field of Statistics, Regression Analysis deals with deriving the relation between a dependent variable and a set of one or more independent variables in a given set of observations. Thus, provided data on observed values for the independent and dependent variables in a problem set, one's objective is to find a function that best predicts the output given the input variables, both for the observed values and for new observations that may be added to the data set.

There are several techniques for performing regression, each applicable in different scenarios. The most typical of them is Linear Regression[1], where the relation between the variables is modeled as a linear combination of the variables and constants. There are also several models for nonlinear regression. However, for all of these, some information about the objective function is assumed: it is either modeled as a linear function, or a polynomial of a specific degree, etc.

The problem of Symbolic Regression refers to a regression model where no information about the structure of the function is assumed. Thus, it is up to the model to infer not only the relationship between the variables but also the *structure* of such relationship. This increases the complexity of the problem significantly, which makes it an attractive candidate for the application of heuristic techniques.

In this paper, we propose an approach towards performing Symbolic Regression modeling it as an instance of Genetic Programming [2]. In Genetic Programming, individuals are modeled as tree structures, where each node represents either a terminal or a function of other nodes. The trees are then altered throughout generations to attempt to improve their fitness towards the objec-

tive function. In section 2 we describe in greater detail the modeling adopted in our approach and the way such modeling was implemented.

2 Problem Modeling and Proposed Solution

In this section we provide a detailed explanation of all the aspects of the Genetic Programming implementation, such as the representation of an individual, the fitness function, etc.

2.1 Individual

As per usual in Genetic Programming approaches, we model individuals as trees, where each node represents either a terminal or a function of other nodes. The function a given individual represents can then be inferred by simply performing an in-order traversal of the tree, where the leaf nodes are terminals and non-leaf nodes are functions. Figure 1 shows an example of an individual that follows this modeling.

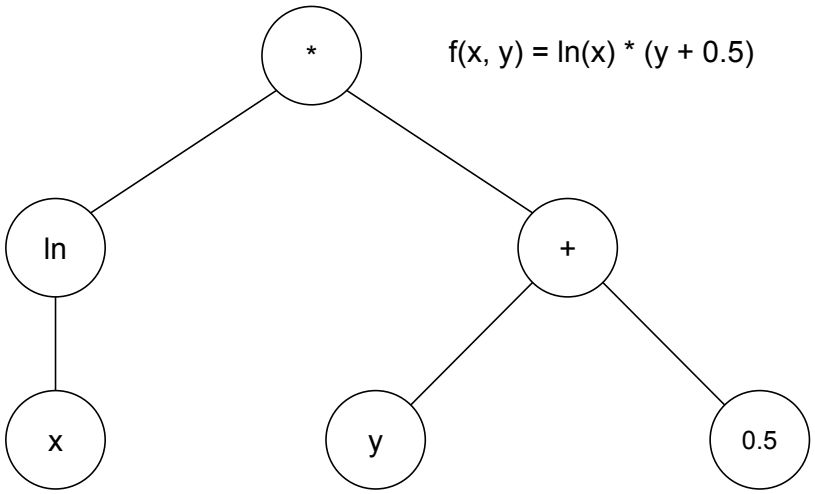

Figure 1: Example of individual in the Genetic Programming model.

Given the data set of observations provided, we chose five possible functions that nodes may represent, of which four are binary and one is unary. They are:

- Addition - Addition nodes evaluate to the sum of both their children.
- Substraction - Substraction nodes evalute to the substraction of their left child by the right child (in this specific order).
- Multiplication - Multiplication nodes evaluate to the product of both their children.
- Division - Division nodes evaluate to the division of their left child by their right child (in this specific order).

- Logarithm - Logarithm nodes evaluate to the natural logarithm (base e) of their child.

The probability of any given function node being assigned to any of the five functions is uniform. That is, $p(+) = p(-) = p(*) = p(/) = p(ln) = 20\%$. To guarantee the soundness of the evaluations, a few exceptions pertaining to the domain of the functions had to be handled. Particularly, when the denominator of a division operation would evaluate to 0, the division itself was made to evaluate to a large integer $(100,000)$. Similarly, for the logarithm operation, when the operand was either negative or zero, the logarithm itself was made to evaluate to a large negative integer $(-100,000)$.

As for terminals, they can fall within two categories:

- Variables - Reference to one of the independent variables in the observation dataset. I.e. $x_i \in \{x_1, x_2, ..., x_n\}$
- Constant - A real number within the range $[-1.0, 1.0]$

Given that variables tend to feature more in functions than constants, the probability of a given terminal node being assigned to a variable is 50% larger than being assigned to a constant. That is, $p(var) = 60\%$ and $p(con) = 40\%$. Constant values are randomly generated within the $[-1.0, 1.0]$ range uniformly.

Therefore, our genotypes can be seen as the set of terminal and function nodes that make up a tree. A genotype's corresponding phenotype is simply the value to which the function it represents evaluates, given a value for each independent variable in the model.

2.2 Initial Population

To promote diversity, the initial population is generated by creating random individuals up until the population size parameter provided. The methodology to generate random individuals follows a very simple algorithm: At every step, the algorithm either generates a function node, which will continue to spawn children recursively, or generates a terminal node, which halts the generation in that subtree. The probability of generating a function node at every step is $p(f) = 0.70^h$, where h refers to the height of the current node. Root nodes have height 0, thus, $p(f) = 0.70^0 = 100\%$, which guarantees that root nodes will always be non-terminals. This method was used to keep the average height of initial individuals reasonably small, without prohibiting a few larger individuals (who may be beneficial) from being generated.

Figure 2 shows how the probability of a given level of the tree generating non-terminals progresses as the tree grows. As one can see, this provides a simple way to generate individuals of varying sizes, which is fundamental to the potential improvement of crossovers in future generations.

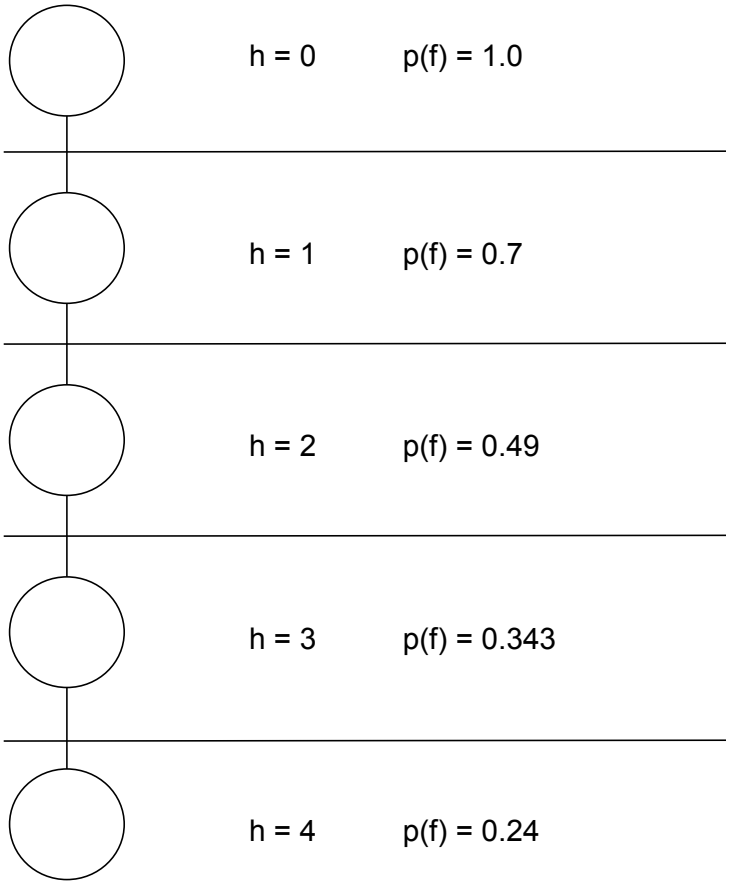

Figure 2: Probability of generating a non-terminal node for each height level in the tree.

2.3 Fitness

Given that our objective is to find the function that best fits all observations, our measure of fitness should reflect how far from the observation a given individual falls. The most natural metrics towards measuring this are residual metrics in regression analysis.

The specific metric chosen in our approach was the Normalized Root-mean-square-deviation (NRMSD). Its formula in our individual evaluation is the following:

$$fitness(Ind) = \sqrt{\frac{\sum_{i=1}^{N}(y_i - eval(Ind, x_i))^2}{\sum_{i=1}^{N}(y_i - \overline{Y})^2}}$$

Where Ind is the individual being evaluated, N is the number of observations, x_i is the set of independent variables for observation i, y_i is the expected outcome for observation i, $eval()$ is the function that evaluates an individual for a given input, and $\overline{Y}$ is the average of the vector of expected outcomes.

It is important to note that, since our fitness function is the value of the residual error, our problem is a minimization problem. The residual is the objective function to be minimized, and a fitness value of 0 implies a function that perfectly fits the data.

2.4 Progressing Generations

Alongside the creation of the initial population, the manner in which new generations are conceived is fundamental to the workings of a Genetic Programming algorithm. In our approach, the current population first goes through a selection step, where a portion of the individuals are selected to be the starting point for the following generation. This generation is then created by iteratively applying our genetic operators on these individuals to create variations of them, so as to repopulate the generation up to the previous one's size. The following sections describe these processes in deeper detail.

2.4.1 Selection

The method of selection in our approach is through tournaments. The size of the tournaments is controlled by a parameter k.

Given a tournament size, k individuals are uniformly randomly selected from the population, and evaluated. The individual with the highest fitness amongst them wins the tournament, and is chosen to be part of the next generation, whilst the others are discarded immediately. This means once an individual loses a single tournament they are already extinguished from the population. While this can potentially eliminate promising individuals — say, if two very-high fitness individuals are randomly picked for the same tournament — it also promotes diversity, which is the main reason why we chose to implement it this way.

Once the tournament selection process is over, $\frac{n}{k}$ individuals have been selected. Therefore, at this point the new generation has size $\frac{n}{k}$. To repopulate it up to size n, we iteratively apply our genetic operators of mutation and crossover to random individuals chosen in the tournament. The probability of whether a mutation or a crossover will occur is also controlled by an input parameter.

2.4.2 Mutation

The simpler of our two genetic operators is mutation. When a mutation is to be performed, a single individual is picked at random from the ones outputted by the selection step. Then, a node from said individual is randomly selected as the mutation point. The mutation process then simply replaces this node by a new randomly-generated subtree.

Note that the same method for generating new trees is applied in this process. Therefore, the same limitations apply, namely that nodes with greater height values have reduced probability of generating subsequent subtrees. Thus, mutation promotes diversity while still avoiding overfitting from the creation of too-large individuals.

Figure 3 shows an example of an application of the mutation operator.

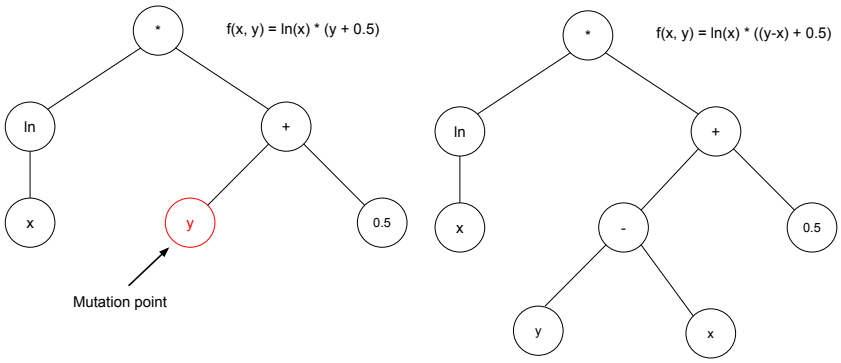

Figure 3: Example of application of the mutation operator to an individual.

2.4.3 Crossover

Whilst mutation is a unary operator, the crossover operator is binary. Thus, once a crossover is chosen to take place two individuals are chosen from the ones outputted by the selection process.

Given the two individuals, the crossover process begins by choosing a random node from the first individual as the insertion point for the crossover. Then, a random node from the second individual is chosen as the appending subtree. Finally, the subtree chosen from the second individual is appended to the first individual, replacing the node chosen as the insertion point. Therefore, a single new individual is created, with mixed characteristics from both parents.

Figure 4 shows an example of a crossover operation.

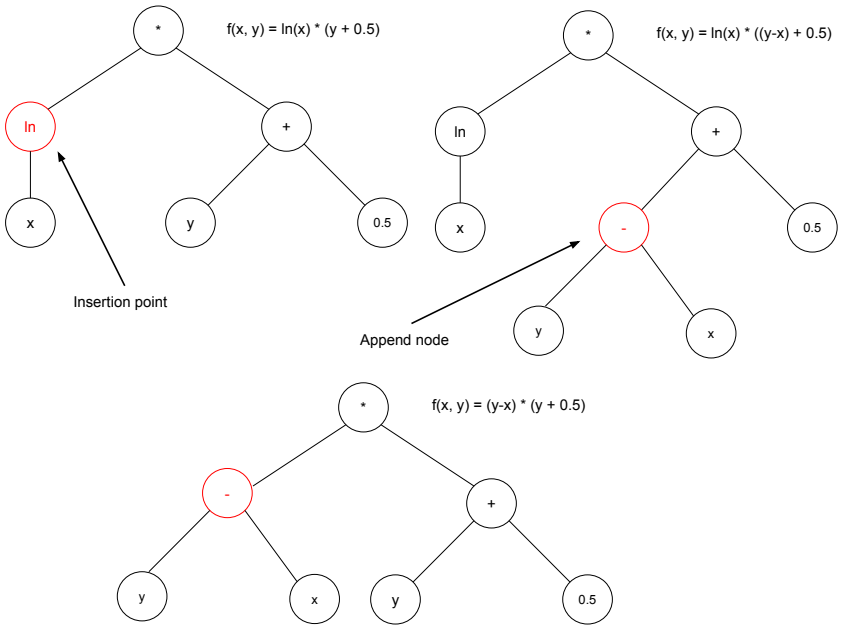

Figure 4: Example of application of the crossover operator to two individuals.

It is important to note that for both operators, elitist versions of them were also implemented. When using elitist operators, the children generated are only included in the new population if their fitness is better than their parents'. Otherwise, the parent with the best fitness is copied over to the next generation instead.

2.5 Convergence

In Genetic Programming, the point at which the algorithm halts can be determined in different ways. A typical approach is based on the fitness of the best individual: once the fitness reaches a certain threshold deemed satisfactory, the evolution can stop and that individual is returned as the best solution. In our context, given the complexity of Symbolic Regression and the uncertainty of whether a reasonable approximation can even be found given real parameters, we chose not to use fitness as a halting criterion. Instead, we execute for a fixed number of generations, which is provided as an input parameter. Once that number has been reached, the best individual is returned as the solution, regardless of its fitness value.

3 Implementation

The proposed solution was implemented fully in C. The main components of the implementation include the tree data structure itself, the handling of input/output, and the implementation of the Genetic Programming algorithm logic. The following sections give an overview of the major factors in the implementation.

3.1 Tree

The tree data structure is one of the cornerstones of the Genetic Programming algorithm, encapsulating most of the information about individuals and how they are structured. The tree is binary and is implemented recursively, so each node in itself is a tree. A node stores the following information:

- **Type** - Stores the node's type amongst Operator, Variable, Constant
- **Height** - The node's height in its tree
- **Left/Right** - Pointers to its left and right children, respectively
- **Data** - Semantic data for the node

Given the fact that node types are mutually exclusive (a node can only be of one type), the semantic data for nodes is stored in a union type, so as to save memory in the node representation. This union type has the following fields:

- **Operator** - For operator nodes, stores which operator it represents.
- **Variable** - For variable nodes, stores which variable in the input it represents.
- **Value** - For constant nodes, store the floating-point value for the real number it represents.

Our tree representation for the individuals also involves a few major operations that nodes support, these are:

- **build_tree**(cur_height, num_vars) - Creates a new tree/subtree starting at the given height.
- **mutate_tree**(node) - Creates a new tree by mutating the input **node**.
- **cross_trees**(node1, node2) - Creates a new tree by performing a crossover between inputs **node1** and **node2**.
- **eval_tree**(node, vars) - Evaluates the tree for the variable values **vars**.

3.2 Genetic Programming

The logic of the Genetic Programming itself is implemented in a separate component. This part of the code interacts with the tree structure strictly through its interface, and does not manipulate individuals directly.

The major functions that implement the algorithm's logic are the following:

- **init_pop**(size, num_vars, seed) - Creates an initial population of the size provided.
- **eval_pop**(pop, size, vars, gen_num) - Evaluates the given population, calculating the fitness of all its individuals.
- **gen_pop**(pop, size, params[]) - Generates a new population, based on the previous one. Implements the fundamental logic of selection and application of genetic operators.
- **calc_fitness**(node, vars) - Calculates the fitness of a single individual for the variable values provided.

4 Compiling and Executing

4.1 Folder Structure

All of the code for the project is available in the root folder. Below is the list of files and a brief description of each:

- tree.c/h - Implement the tree data structure and its interface.
- gen.c/h - Implement the Genetic Programming algorithm and its interface.
- main.c - Processes input and kickstarts algorithm execution.
- run.sh - Bash script for running the algorithm with a few default parameters.
- Makefile - Defines compilation targets for the project.

4.2 Compiling

A Makefile is provided within the root folder. Therefore, to compile the program one can simply run "make". A binary called "gen" should be generated in the same folder, which can then be used to run the algorithm.

4.3 Running

To run the program, the command format is:

```
$ ./gen <train_input> <test_input> <seed> <pop_size>
<num_gen> <elit_flag> <cross_prob> <tourn_size>
```

Where the parameters are:

- train_input - Input file with training data.
- test_input - Input file with test data.
- seed - Value for the random number generator seed.
- pop_size - Size of the population.
- num_gen - Number of generations to run for.
- elit_flag - Determines the use of elitist operators.
- cross_prob - Determines probability of performing crossover.
- tourn_size - Determines the size of the tournaments during selection.

The Makefile provided includes a target for running the program. To do so, one can simply type "make run" after having compiled the program. The parameters used for these runs are to reproduce the very last experiment described in this report, where the concrete dataset was tested with 250 generations. The program should run once for each random number generator seed used. Also provided is a bash script called "run.sh", which contains the parameters for all parameter tuning experiments performed, in order. Running this script should run all of the experiments described in subsection 5.2.

4.4 Output

The program outputs to different streams. To the standard output (stdout), it prints the statistics for the evaluation of each generation. These include average fitness, worst fitness, best fitness, number of repeated individuals, etc. It also prints the evaluation of the last generation for the test input, showing its best/worst/average fitness. To the standard error output (stderr), it prints lots of data regarding the execution, such as the individuals in each population, their fitness/height, data regarding the selection process, etc.

5 Experimentation

5.1 Methodology

All the experiments were run using some of the three datasets provided, of which two were synthetic and one concrete. The hardware used features an Intel i5 4670K at 3.8 GHz, with 16 GB of RAM. The program however was not parallelized, and therefore runs in only one core. The software used was the Windows Subsystem for Linux, for running Linux distros within Windows 10.

For each experiment, five instances of the execution with the same parameters were performed, each using a different seed for the random number generator. Then, the values reported for each execution are the mean amongst all executions combined. While only five executions may seem like a small number, the standard deviation for all reported metrics was computed and found to be reasonably small, which justifies the fewer number of runs, since extra executions would be unlikely to reduce this much further.

5.2 Parameter Choice

The goal of the first stage of experimentation was to empirically choose the best parameters for execution. To this end, several parameters were tested in order of seeming impact on algorithm performance.

5.2.1 Population Size and Number of Generations

The two most important parameters are the size of the populations and the number of generations. This is due to the impact the initial population has on the percentage of the solution space that will be explored, and the amount of exploration/exploitation that can be performed during the limited number of generations. Given that one might affect the other, it is unlikely that they could be chosen independently. Therefore, we ran experiments for all combinations of three values for each: 50, 100, and 500. Through the analysis of the performance of each combination we can infer the one that provides the best results. While testing for these parameters, all the other parameters were kept constant. Particularly, the tournament size was set to **2**, the probability of crossover to **0.9**, and elitism was set to **true**. This first set of tests was performed in the first synthetic dataset provided (synth1).

It is important to note that in these graphs a maximum threshold of 1.5 was imposed to the fitness values displayed. This is necessary because in early generations the values of average and worst fitness could be distorted by some

individuals which caused the chart's scale to be deformed. Nevertheless, we stress that for all cases the values eventually fell below the 1.5 threshold range within at most 10 generations. Therefore, no generality is lost when limiting the data range.

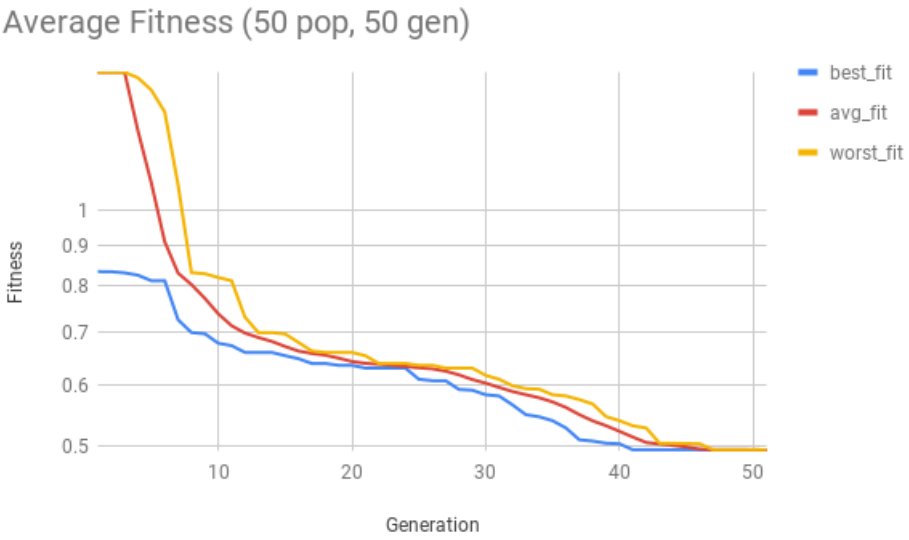

Figure 5: Average fitness for population size 50 and 50 generations.

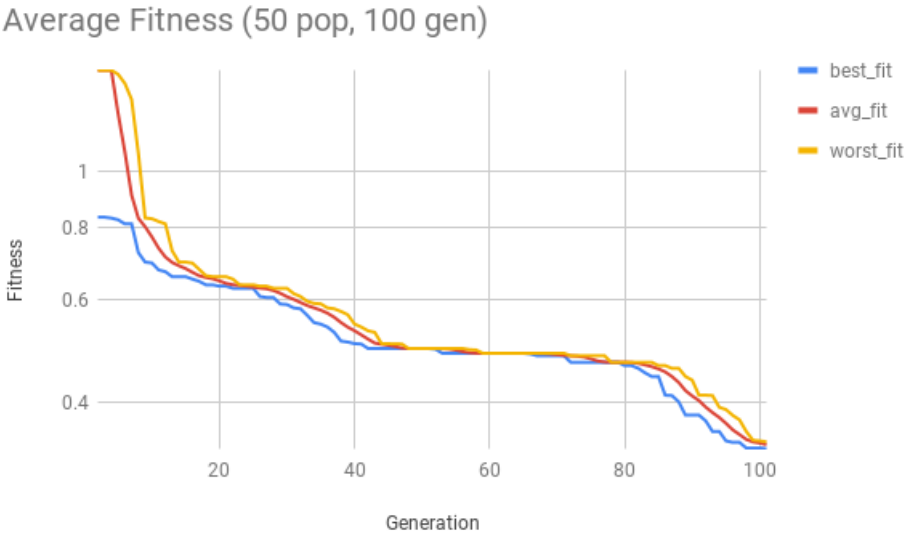

Figure 6: Average fitness for population size 50 and 100 generations.

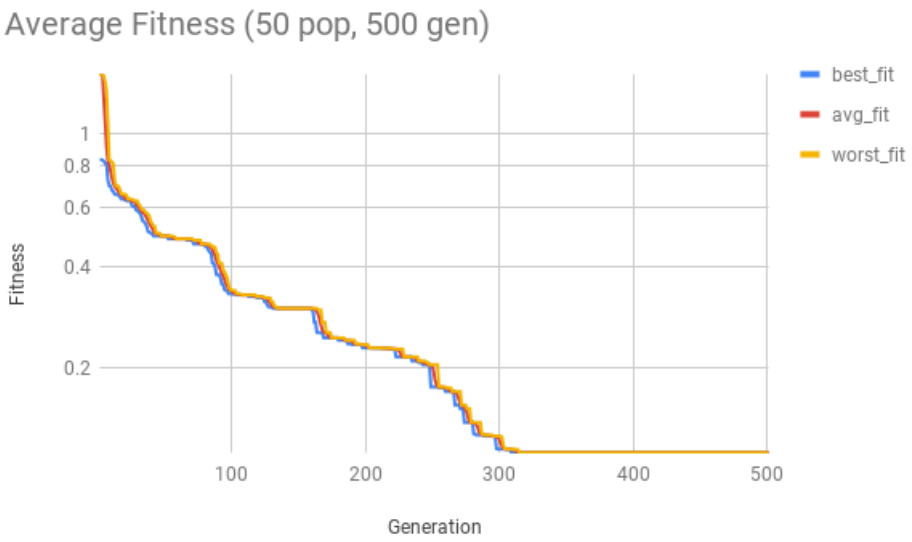

Figure 7: Average fitness for population size 50 and 500 generations.

Figures 5, 6 and 7 show the results for population size 50 and number of generations 50, 100 and 500, respectively. Through the analysis of these charts it is already clear that the number of generations drastically affects the algorithm's performance. While for a number of 50 generations the best fitness value achieved was a little under 0.5, when the number increases to 500 the best and average fitness values reach values close to 0.1, which is a very sizeable improvement. Moreover, it is interesting to note that the population only seems to converge around generation 300. This improvement is also fairly constant, with not many periods of subsequent generations with equal best fitness. This indicates that the algorithm is making steady progress, and likely to be exploring the search space efficiently. However, it also indicates that the initial population does not provide much diversity, and it is up to the process of mutations and crossovers to progressively explore better solutions.

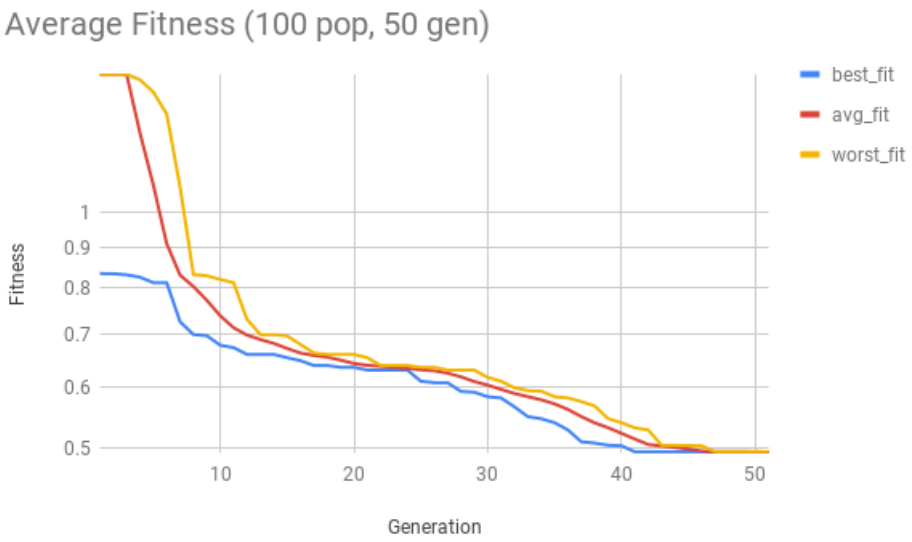

Figure 8: Average fitness for population size 100 and 50 generations.

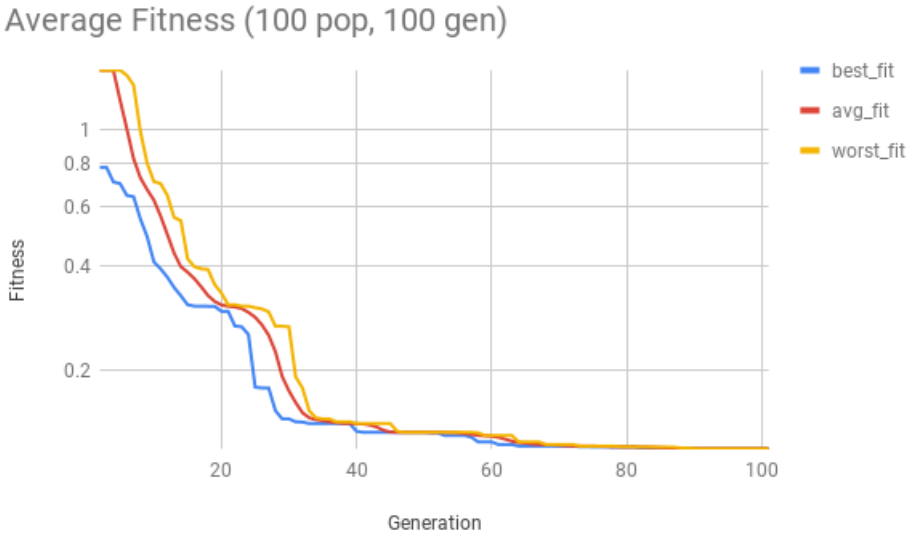

Figure 9: Average fitness for population size 100 and 100 generations.

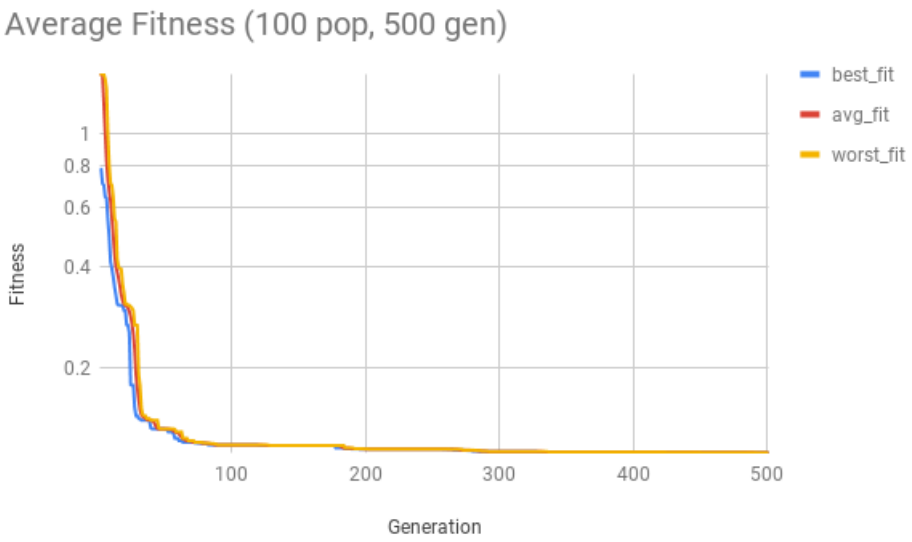

Figure 10: Average fitness for population size 100 and 500 generations.

Figures 8, 9 and 10 show the results for population size 100 and number of generations 50, 100 and 500, respectively. Once again, the algorithm shows progressive and consistent improvement until convergence is reached, and then seems to plateau and hardly ever find improving individuals. However, it is clear to see that doubling the population size causes it to converge much faster. While it took close to 300 generations for it to converge earlier, for a population of 100 individuals it seems to find a near-best solution at around the 70th generation. While some improvement takes place after this point, its profitability may be questionable, when one considers the added runtime. Interestingly, the best fitness found for a population of size 100 is roughly the same as that of size 50. Therefore, it appears there is only a tradeoff between population size and convergence: doubling the population causes convergence to be reached almost 4 times faster, but both approaches converge to almost the same solution.

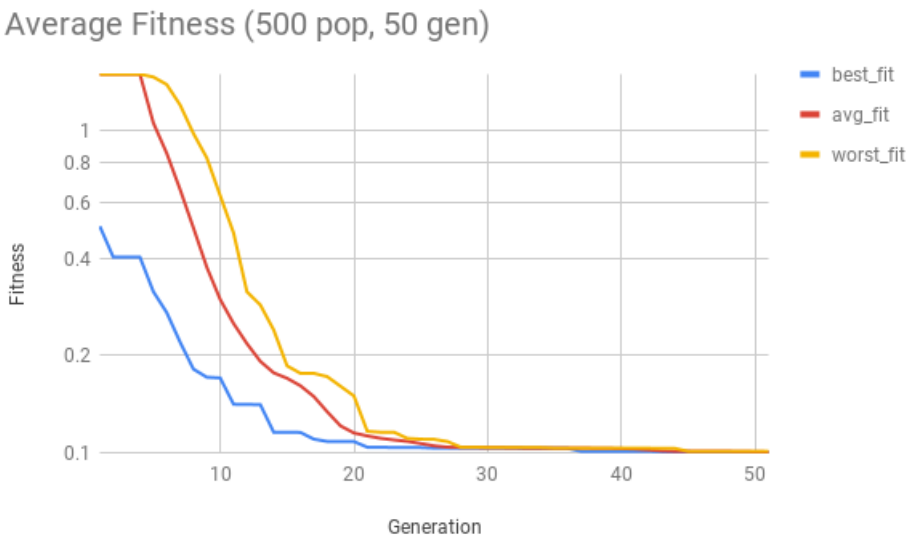

Figure 11: Average fitness for population size 500 and 50 generations.

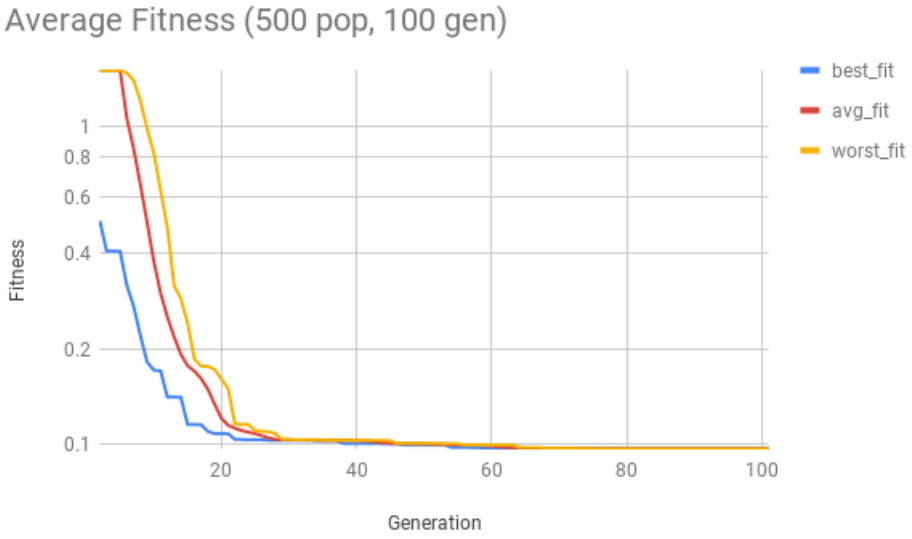

Figure 12: Average fitness for population size 500 and 100 generations.

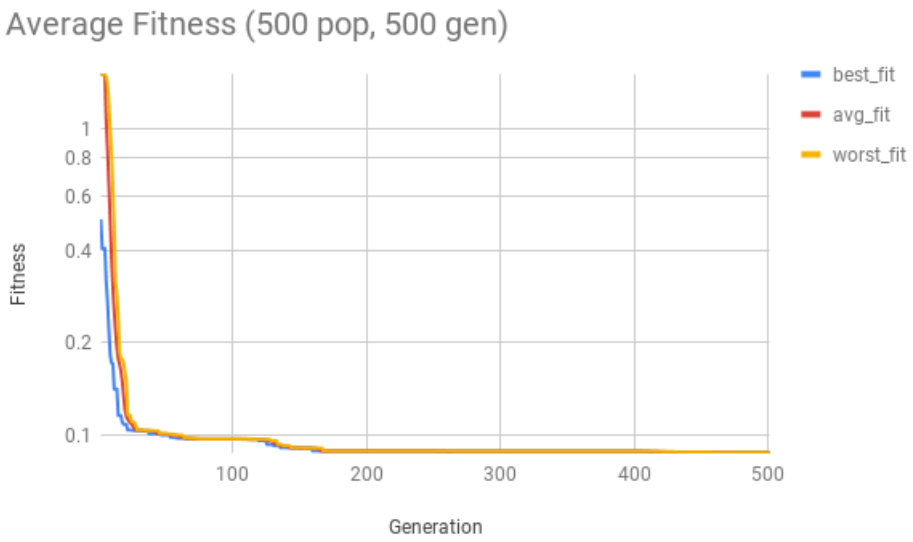

Figure 13: Average fitness for population size 500 and 500 generations.

Figures 11, 12 and 13 show the results for population size 500 and number of generations 50, 100 and 500, respectively. For an initial population of size 500, the algorithm progresses much faster. Before the 30th generation it already reaches solutions of comparable quality to the best solutions found for other population sizes. From that point on, it progresses more slowly, but is still able to improve solutions gradually. This indicates that while significant exploration is achieved at the beginning of execution, there is still some room for improving solutions once a more exploitation-focused behavior kicks in. Interestingly, while it initially seems that the algorithm converges well before the 100 generation mark, it still finds somewhat significant improvements at the 130th generation and 450th generation marks. These improvements, however, are of around 0.01 fitness, and are unlikely to be worth the added onus on the execution time of the algorithm.

Another point of interest regards how the parameters performed for the testing dataset once training was done: for all of the combinations tested, the performance of the best individual in the test dataset was within a 10% margin of its best performance in the training dataset — a strong indicator that the algorithm does not overfit. Therefore, given the analyses performed above and the fact that all combinations generalize well, the ideal parameters for population size and number of generations chosen were $pop_size = 500$ and $num_gen = 100$, as these proved to provide the best tradeoff between quality of solution and CPU time.

5.2.2 Crossover/Mutation Probabilities

Once the population size and number of generations parameters were chosen, we moved on towards tuning the probabilities for crossover and mutation. Intuitively, a higher probability of mutation promotes diversity, whereas crossovers

tend to favor exploitation of already promising individuals. To test the best combination of these, we performed experiments with two additional setups (besides the one with $p_c = 0.9$ from the previous section). These were for values of probability of crossover $p_c = 0.8$ and $p_c = 0.6$ and probability of mutation $p_m = 0.2$ and $p_m = 0.4$, respectively.

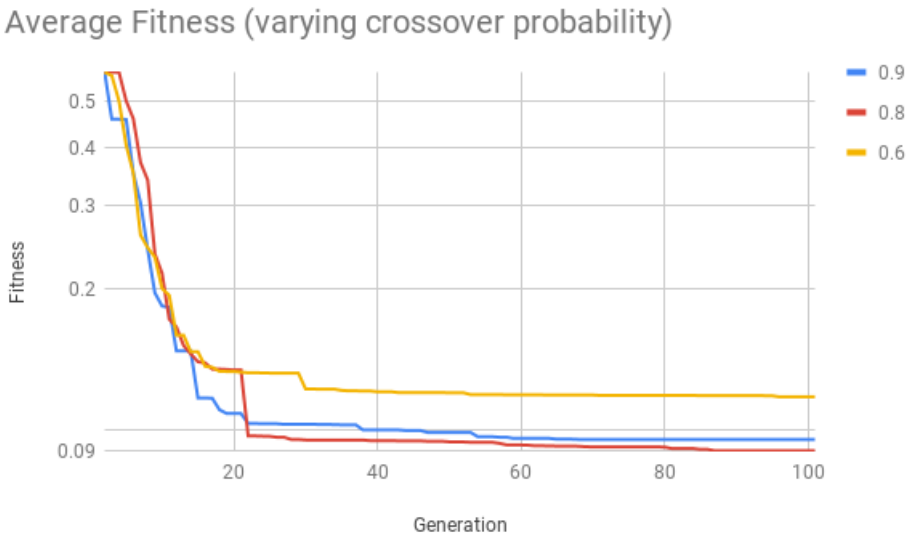

Figure 14: Average fitness for different crossover probability values.

Since the values of worst and average fitness eventually converge to the same as the best fitness, from this point on we only show values for best fitness, to improve the visualization of the data. Figure 14 shows the results for the different crossover probability values. Lowering the probability by a small amount seems to improve performance marginally, with a slight improvement from the earlier setting, as can be noted in the chart with the change from 0.9 to 0.8. This is likely due to the extra diversity provided by the additional mutations that occur. However, lowering the probability too much eventually produces worse results. This is due to the algorithm then focusing too much on diversity and never exploring promising individuals further, which gimps the fitness of the overall population.

Given the insights obtained from these experiments, the ideal probability parameter was chosen as $p_c = 0.8$, which implies a probability of mutation $p_m = 0.2$.

5.2.3 Tournament Size

The next parameter under scrutiny was the size of the tournament. Intuitively, larger tournament sizes should increase selective pressure, due to lower fitness individuals being more likely to be eliminated from the population. To test this hypothesis, once again three settings were tested: the previous size of **2**, and new sizes **5** and **10**.

Figure 15: Average fitness for different tournament size values.

Figure 15 shows the result of this experiment. The results are actually quite similar to the effect of reducing the crossover probability, with an increase in tournament size initially providing better results, but further increasing the value bringing about a worsening of the population instead. This once again implies a delicate balance between exploration and exploitation. And initial increase enforces selective pressure and focuses on exploitation, by eliminating poor individuals earlier. However, increasing the value too much causes the algorithm to lose generality by not exploring enough of the solution space. Given this analysis, the ideal parameter for tournament size was set as $k = 5$.

5.2.4 Elitism

Finally, the effect of elitism in the application of the genetic operators was evaluated. Up until this point, all experiments were made using elitist operators, under the assumption that always keeping the best individuals will eventually lead to better solutions. However, it is possible that the added diversity from occasionally eliminating good individuals may bring about a fitter population in the end. To test this, we performed an experiment with elitist operators disabled.

There are two caveats to this experiment: first, the hardware/software for this experiment had to be changed, due to the previous computer failing (which also helps explain the delay in turning in this assignment). Therefore, results for the previous parameter of elitism enabled may also differ, due to different implementations of the random number generator across platforms and whatnot. Nevertheless, we stress that we are comparing two results from experiments performed on the same setting, and therefore the insights should still be valid. Second, due to the nature of non-elitist operators, the worst and average fitness do not converge to the best fitness as happened with the other experiments.

Still we only choose to display the value of the best fitness here, given that it is the most useful metric.

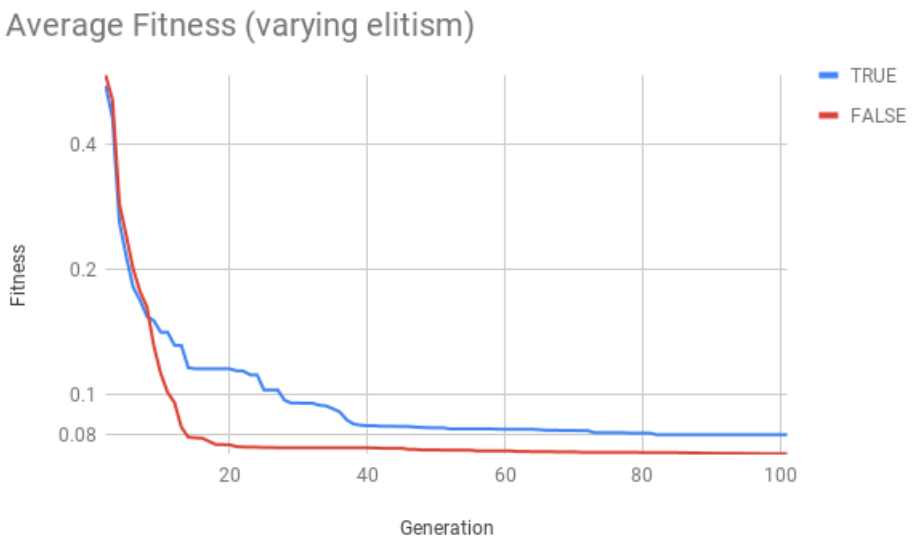

Figure 16: Average fitness when using or not using elitist operators.

Figure 16 shows the result of this experiment. Surprisingly, disabling elitism actually significantly improves the quality of the best solution found. Apparently, the added diversity only slightly worsens the quality of the first few generations, but allows the algorithm to find vastly better individuals early in the selection process. Then, the variety of individuals allows a greater degree of exploration, which eventually causes the algorithm to converge on solutions that are better than when using elitism. Given this somewhat shocking result, the ideal value for elitism was chosen as $elit = False$. Note that the best fitness value still never goes down. This is due to the way the selection process was implemented, where the individuals who win the tournaments always move on to the next generation. This means even if they are picked for mutations/crossovers, they will only potentially generate new individuals, but will also remain in the population themselves.

5.2.5 Evaluation on Second Synthetic Dataset

Once the parameters were chosen, we moved on to evaluating how the found setting performs on a more general scale. Given that we could have caused the algorithm to become too specific towards the dataset being used, we chose to evaluate it on a different dataset, the second synthetic one provided.

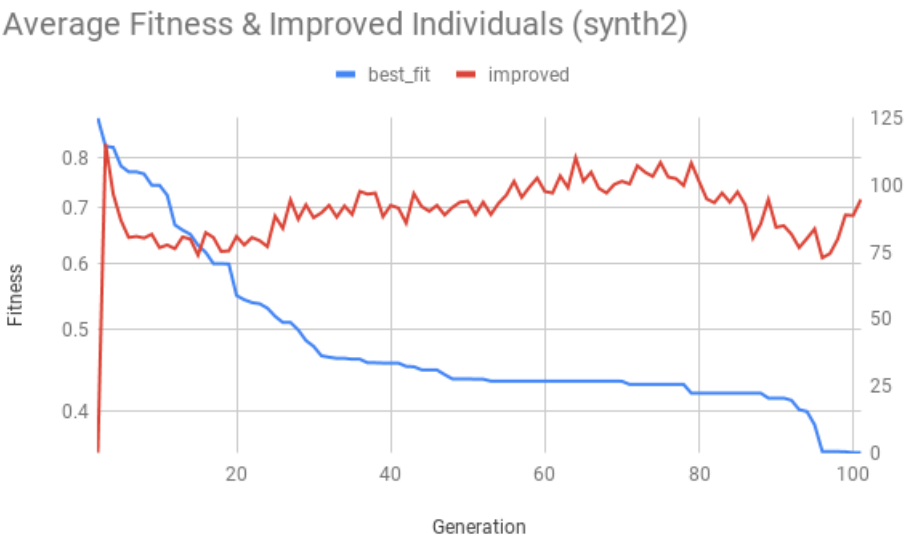

Figure 17: Average fitness and number of improving individuals for the synth2 dataset.

Figure 17 shows the results for this first experiment, with the average best fitness and number of improved individuals created per generation. The number of improving individuals generated remaining high throughout is not surprising, given that we are not using elitist operators. What is suprising, however, is that while the algorithm does improve over time, the fitness value for the new dataset was much worse than the one seen in the previous dataset. Not only this, but while the test fitness of the best solution in the previous dataset was within 10% of the training fitness, in this case this difference went up to 121%. This indicates one of two things: either our algorithm became too specific for the first dataset, and tends to not generalize well (overfit) for other datasets, or the second dataset may be more complex, and the algorithm did not have enough time to learn better fitting individuals.

To test this second hypothesis, we first observed the results more carefully. The fitness over time seems to indicate that even during the last generations the algorithm is still making progress. This indicates that it may not have converged as fast as it did for the previous dataset, which corroborates the theory that the new dataset may simply be more complex. Therefore, we decided to readjust the parameter for number of generations, running the algorithm with 250 and 500 generations instead, to see the effect on the performance.

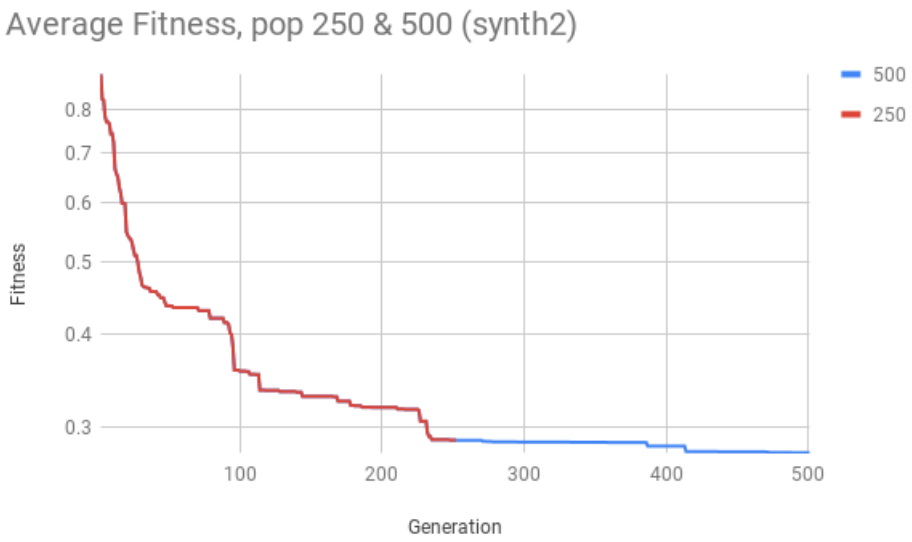

Figure 18: Average fitness for population sizes 250 and 500 in the synth2 dataset.

Figure 18 shows the result of this experiment. Indeed, the fitness values for the other population sizes indicate that the algorithm still does improve significantly in further generations, and actually progresses up to the final few generations, eventually reaching fitness values of 0.25, a very big improvement. Unfortunately, when evaluated on the test inputs, the discrepancy between test and training fitness increased from the earlier 121% to 1000% for 250 generations and 2000% for 500 generations, which indicates a very sharp degree of overfitting. Therefore, given that overfitting is a much bigger problem than worse values of fitness, we decided to keep the parameters as they were.

5.2.6 Concrete Dataset

Finally, after all the parameter tuning had been performed, we tested the algorithm's performance for the concrete dataset provided. Figure 19 shows the result for this experiment. Once again, the average fitness values observed were much worse than the ones for the first synthetic dataset, once again indicating that the algorithm is worse at predicting more complex functions. Not only that, but once again the algorithm seemed to not have converged with only the 100 generations provided. However, in this case the discrepancy between test and training performance was close to 10%, rather than the 121% from the synthetic dataset. Therefore, we conjectured that while there was room for more learning in future generations, the algorithm was not yet overfitting. Thus, we decided to run the same experiment on the concrete dataset, varying the number of generations to 250 and 500.

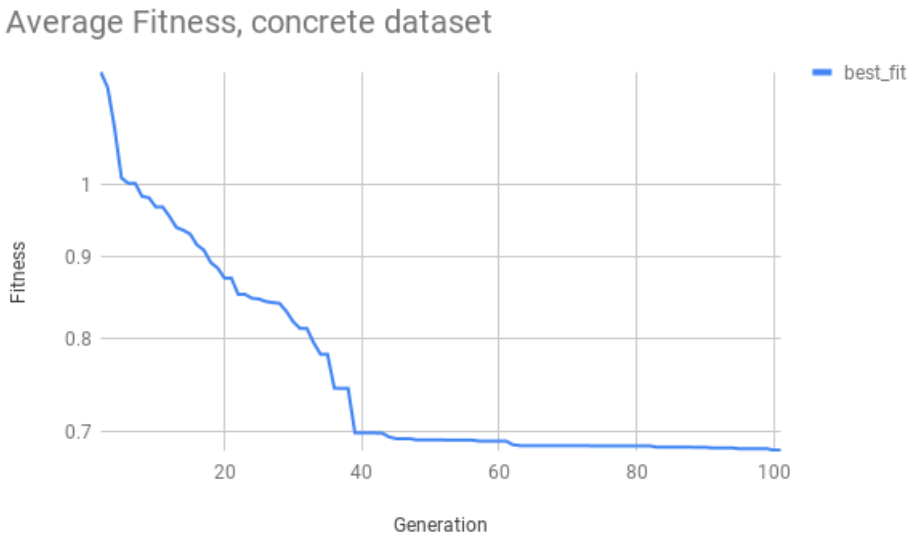

Figure 19: Average fitness for population size 100 in the concrete dataset.

Figure 20 shows the result of this experiment. The chart shows that indeed there is much room for improvement when adding new generations. In fact, around the 150th generation there is a sharp improvement in the overall quality of individuals. After the 300th generation several steps of improvement also occur, until eventually the average fitness reaches values of around 0.58, a vast improvement over the previous best. However, given that what truly matters is the testing data fitness, we analyzed those as well. For 250 generations, the improvement in test fitness follows the one seen in training, with the gap between training and testing remaining consistent with previous experiences. Therefore, the increase to 250 generations improves the algorithm's performance significantly. However, when increasing to 500 generations, while the training fitness also improves somewhat, the testing data fitness actually decreases significantly, which indicates that the algorithm may be beginning to suffer from overfitting. Therfore, the ideal parameter is most likely around the 250 generations mark.

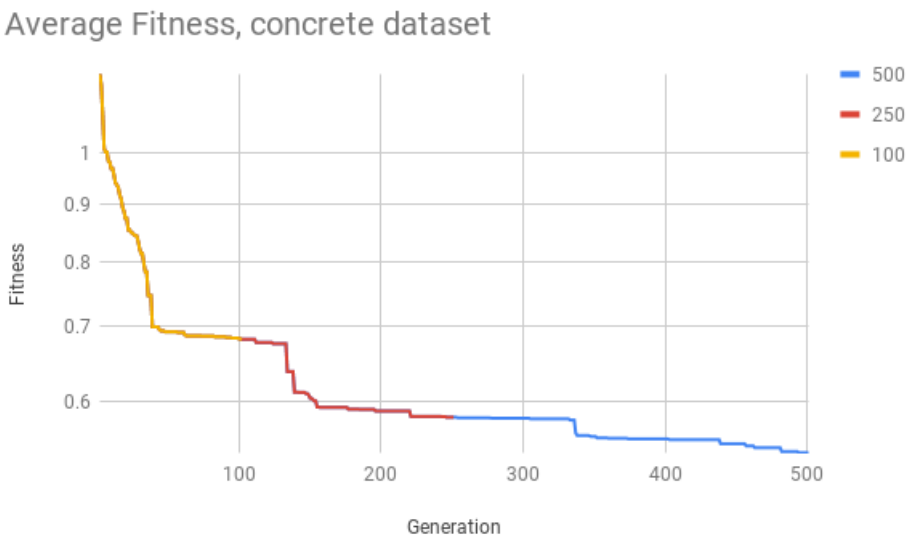

Figure 20: Average fitness for population size 100 in the concrete dataset.

References

[1] Neter, John, et al. Applied linear statistical models. Vol. 4. Chicago: Irwin, 1996.

[2] Koza, John R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*