Structure-Based Genetic Programming Prediction for Bike Trip Duration

COS 710 - Artificial Intelligence I

Isheanesu Joseph Dzingirai

April 2023

1 Introduction

Structure-based genetic programming (SBGP) is an evolutionary algorithm and a variant of genetic programming that is used to solve problems in artificial intelligence, machine learning, and optimization. SBGP is specifically designed to handle problems where the structure of the solution is critical to its success.

In SBGP, populations of programs evolve through processes of selection, crossover, and mutation inspired by the principles of natural selection and genetics. The main difference from traditional genetic programming is that the SBGP algorithm emphasizes the structural features of programs during the evolutionary process.

The report explores the use of SBGP for predicting the duration of bike trips based on various input features such as weather conditions, distance travelled, and time of day, and output an estimated duration for a bike trip. It also discusses the data used for training and testing , the methodology used for the SGBP process, and the performance of the evolved program in predicting bike trip duration compared to other models including the standard genetic programming model. Overall, this project aims to demonstrate the effectiveness of SBGP over the canonical GP for predicting bike trip duration.

2 Experimental Setup

2.1 Programming Language and Libraries

The genetic program was implemented using Python 3.8.10 with the following libraries:

- pyarrow - library for efficient data interchange between Python and other languages such as Rust which provides tools for serialization, and memory-efficient data structures.
- pandas - data analysis and manipulation library that provides easy-to-use data structures and data analysis tools for data cleaning, merging, filtering, and transformation.
- numpy - numerical computing library that provides fast and efficient tools for numerical operations on large arrays and matrices.
- numba - just-in-time (JIT) compiler which compiles Python code into machine code at runtime, allowing for significant speed improvements for numerical calculations and scientific computing.
- sklearn - machine learning library that provides simple and efficient tools for data mining and data analysis, including regression, and dimensionality reduction algorithms.
- scipy - scientific computing library that provides tools for optimization and more.

2.2 System

The experiments were performed on a Google Colab with the following machine specifications:

RAM :	12.7 GB
Disk Space:	78.2 GB
GPU:	NVIDIA TESLA T4
CPU:	Intel(R) Xeon(R) CPU @ 2.20GHz

2.3 Structure-Based Genetic Program Parameters

The following parameters were used to tune the genetic program side of the model:

Fitness Function:	$MedAE$
Population Size:	50
Number of Features to Use:	10
Train Size:	0.7
Test Size:	0.3
Random State:	42
Function Set:	$+, -, /, *, sqrt$
Terminal Set:	a, b, c, d, e, f, g
Initial Tree Depth:	6
Maximum Tree Depth:	8
Tree Generation Method:	$RAMPED$
Tournament Size:	10
Crossover Rate:	0.7
Mutation Rate:	0.3
Maximum Generations:	50

The following parameters were used to tune the iterative structure based (ISBA) side of the model:

• m:	10
• n:	5
• d:	3
• rcomp:	$NODE$
• rthresh:	0.5
• gcomp:	$NODE$
• gthresh:	0.5
• lthresh:	3

The meaning of the parameters are: m is the number of global level runs, n is number of local level runs, d is depth until which a component is fixed, *rcomp* is the type of root comparison for gsim, *rthresh* is the maximum number of local optima which an individual can have the same (or same type of) root as, *gcomp* is the type of global area comparison for gsim, *gthresh* is the maximum number of nodes, from the root to depth d, which an individual can have that is the same (or the same type) in the same position in one of the local optima, *lthresh* is the maximum number of relations that an individual can have in common with one of the local optima.

3 Data description, Pre-processing and Feature Selection

3.1 Data Description

The data that was used to train and test the genetic programming model was obtained from Mendeley Data, which they got the data from a Seoul Bike sharing system and processed it.

The dataset contained the following 24 features:
Distance, PLong, PLatd, DLong, DLatd, Haversine, Pmonth, Pday, Phour, Pmin, PDweek, Dmonth, Dday, Dhour, Dmin, DDweek, Temp, Precip, Wind, Humid, Solar, Snow, GroundTemp, Dust and a target variable **Duration**.

Each of the features' value and target variable value was a numerical value.

3.2 Pre-processing

Once the data was read into a dataset, all rows with null values were removed and outliers were removed by doing the following:

- Calculating a z-score for each feature in the dataset. The formula to calculate the z-score was $z = \frac{x-\mu}{\sigma}$ where x is the value of row in the column, μ was the mean of the column, and σ the standard deviation of the column
- Removing any data point with a z-score greater than 3 or less than -3

3.3 Feature Selection

Feature selection is the process of selecting a subset of relevant features (variables or attributes) from a larger feature set for use in building predictive models or solving machine learning problems.

This was done to to reduce the number of features while retaining as much information as possible, improving model accuracy, interpretability, and generalization..

'SelectKBest', a uni-variate selection method from Sklearn, was used to perform feature selection on our dataset. Each feature was given a score using f-regression which is a statistical metric based on their correlation to the target variable 'Duration'. The top 10 features with the best scores were selected.

The selected features were: **Distance, Haversine, Phour, PDweek, Dhour, DDweek, Temp, Wind, Humid, GroundTemp**

After the feature selection, all the features that were not selected were removed from the dataset.

4 Representation Used and Terminal and Function Sets

4.1 Solution Presentation

A probable solution program was represented using a parse tree.

The reason for this is that a parse tree offers a flexible and hierarchical approach to describe complicated solutions that may be constructed out of simpler parts. Given its ability to represent solutions of different levels of complexity and its ease of manipulation by genetic operators like mutation and crossover, the tree structure is particularly advantageous for SBGP. The tree's nodes can be functions or terminals.

Additionally, the tree structure can be utilized to represent a variety of issues, including symbolic regression in addition to classification and regression tasks. The efficient and effective programs that are generated are simple to perceive and understand.

The tree's terminals were represented by letters of the alphabet that reflected data such as distance traveled, weather conditions, and time of day, whilst the non terminals were mathematical operators such as addition, subtraction, multiplication, and division.

4.2 Function Set

Arithmetic operators +, -, *, /, sqrt were used as the internal nodes of the tree. For the / operator, if the division was by 0, the value 0 was returned. For the sqrt operator, if the value was negative, the sqrt was evaluated on the absolute value.

4.3 Terminal Set

The following letters for the terminal set represented the following features:

- a: Distance
- b: Haversine
- c: Phour
- d: PDWeek
- e: Dhour
- f: DDweek
- g: Temp
- h: Wind
- i: Humid
- j: GroundTemp

5 Methodology:

5.1 Initial Population Generation

The tree generation method was a user-defined parameter *TREE_GENERATION_METHOD.* The value of the parameter could have been 1 of the following:

- RAMPED: For the ramped half-and-half method
- FULL: For the full method
- GROW: For the grow method

It was used to produce a random population of candidate programs with a maximum starting tree depth. The number of candidate programs generated was specified by a user-defined *POPULATION_SIZE* parameter and the maximum starting tree depth was specified by a user-defined *INITIAL_TREE_DEPTH.*

For the model, the ramped half-and-half method was used to generated the random population of candidate programs as it combines both the full and grow methods. At each depth from a depth of 2, half of the candidate programs were generated using the grow method while the other half were generated using the full method. This helped increased the diversity of the population created.

Every candidate that was first generated for the algorithm's global search (exploration) was different, whereas every candidate that was initially generated for its local search (exploitation) was identical from the root to a depth d.

5.2 Fitness Function and Fitness Evaluation

The genetic program's fitness functions were the Root Mean Squared Error (RSME), Median Absolute Error (MedAE), and Mean Absolute Error (MAE) between the anticipated and actual cycling ride duration.

Each of the fitness functions was applied to each candidate program in the population of candidate solutions for fitness evaluation. This entailed running each program on a set of train or test data and comparing the output of the program to the expected output for each case.

This was done to guide the structure-based genetic program model in selecting the fittest programs for reproduction and evolution.

5.2.1 Root Mean Squared Error (RMSE):

RMSE is a measure of the difference between the predicted and actual values of the dependent variable. It is calculated by taking the square root of the average squared difference between the predicted and actual values.

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

Where y_i is the observed value, $\hat{y}_i$ is the predicted value, and n is the number of observations. The square of the difference between the observed and predicted values is summed over all observations, then divided by the total number of observations. Finally, the square root is taken to obtain the RMSE.

The advantage of using RMSE as a model performance statistic is that it pays more weight to larger errors. RMSE is more sensitive to larger errors in predictions since it takes the square root of the average of the squared discrepancies between expected and actual values. This means that if a model has a few extremely significant errors, the RMSE number will reflect this, and the model will be penalized as a result.

Overall, its sensitivity to greater errors is also a helpful attribute, making it especially effective in situations when huge errors are highly costly.

5.2.2 Median Absolute Error (MedAE):

MedAE, or Median Absolute Error, is a statistical metric used to assess the accuracy of a prediction model, such as a regression model. It is a robust scale measure, which means it is less affected by extreme values or outliers in the data.

First, we determine the absolute error between the predicted and actual values for each observation in the dataset to compute the MedAE. The median of these absolute errors is then calculated. In other words, the absolute errors are sorted from least to largest, and the median number is chosen as the MedAE.

$$MedAE = median(|y_i - \hat{y}_i|) \tag{2}$$

where y_i is the actual value and $\hat{y}_i$ is the predicted value for observation i.

MedAE has the advantage of being less susceptible to outliers than other error measures such as Mean Absolute Error (MAE) or Root Mean Squared Error (RMSE). This is due to the fact that the median is a robust measure of central tendency, which means it is less influenced by outliers in the data.

Overall, MedAE is a valuable error measure to employ when working with datasets including outliers or extreme values. It allows for a more robust evaluation of model performance and can aid in the identification of models that are more suited to dealing with such datasets.

5.2.3 Mean Absolute Error (MAE):

MAE, or Mean Absolute Error, is a statistical metric that measures the average magnitude of the mistakes in a regression model between predicted and actual values. It is calculated by taking the absolute difference between the predicted and actual values and averaging them throughout the whole dataset.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}| \tag{3}$$

where n is the number of observations, y_i is the actual value, $\hat{y_i}$ is the predicted value, and $|\cdot|$ represents the absolute value.

MAE has the advantage of being simple to interpret. Because MAE is a measure of the absolute difference between expected and actual values, it provides a clear indicator of how far off the average predictions are. Furthermore, because MAE is determined using the absolute value of the error, it is less sensitive to outliers than other metrics that square the error, such as mean squared error (MSE).

Overall, MAE is a valuable statistic for evaluating the accuracy of regression models, particularly when the dataset contains outliers or when an easy-to-interpret measure of error is required.

5.3 Selection Method

The selection method used was tournament selection. In tournament selection, a fixed number of individuals t are randomly selected from the population and the individual with the best fitness from the t individuals is chosen as a parent for the next generation.

The reason this selection method was chosen over fitness proportionate selection is that fitness proportionate selection may lead to the SBGP experiencing premature convergence if the population of the SBGP contains individuals with a very high fitness.

Tournament selection also has the following advantages:

- Tournament selection can assist maintain diversity within the population by randomly selecting individuals for each tournament, preventing early convergence to a suboptimal solution.
- By allowing weaker people to be selected as parents on occasion, tournament selection can assist balance exploration and exploitation of the search space, preventing the algorithm from becoming locked in a local optimum.

5.4 Genetic Operators

5.4.1 Crossover

The two parents chosen through tournament selection were subjected to the subtree crossover operator to create two new offspring individuals. To create the two new offspring individuals, the operator chose a random sub-tree (a section of the tree that is a self-contained substructure) from each parent individual.

The offspring were examined to make sure that their tree depth did not go beyond the permitted tree depth. When an offspring's tree's depth exceeded the allowed depth, it was pruned until its depth was the same as the allowed depth.

The rate at which the sub-tree crossover operator was applied was high in order to prevent premature convergence of the genetic program by exploring different areas of the program space.

5.4.2 Mutation

The two parents selected through the tournament selection process were then perhaps put to crossover and the mutation operator.

The following actions led to the mutation of each of the two trees:

- Selecting a random mutation point in the tree that is not the root node.
- Generating a subtree using the full or grow methods, with a depth between one and the difference between the level of the node of the mutation point and the maximum depth of the tree.
- Using the recently created subtree to take the place of the mutation point.

The use of this operator was necessary to guarantee that the offspring were sufficiently different from both their parents and other population members. In order to keep the genetic program from becoming too chaotic while still allowing for enough variety to successfully explore the program space, the mutation operator's rate was kept modest.

6 Similarity Indexes

6.1 gsim

gsim function checks whether the number of local optima that have the same value as a given node and the number of local optima that match the node up to a given depth to evaluate whether a node is sufficiently similar to a list of local optima. The function outputs true if the program is sufficiently similar and false otherwise.

In the model, it was implemented as follows:

- The function took a single node, a list of local optima (other nodes representing the best solution in a given region), a root threshold, a component threshold, and a depth. define. The function returned a boolean value indicating whether the candidate was close enough to the local optimum based on these thresholds.

- The function first checked if the local optima list is empty. If so, the function returned false, indicating that the individual does not resemble a local optimum.

- The function then counted the number of local optima that have the same value as a single node. If the ratio of these local optima to the total number of local optima was greater than the root threshold (rthresh), the function returned true, indicating that the individual is close enough to the local optimum.

- If the root threshold condition was not met, the function called an internal helper function which took two nodes representing the tree and a depth ('length') define This helper function recursively compares two trees up to a specified depth by checking if the values of the nodes at each level match. The helper function returned true if the tree matches the specified depth otherwise it returned false. If the ratio of these local optima to the total number of local optima was greater than the component threshold (gthresh), the function returned true, indicating that the individual is close enough to the local optimum.

6.2 lsim

The lsim index is used to detect similar individuals during local-level searches when similarity verification is required. The index compares people by calculating the number of relationships shared by both individuals. A relation is defined as a tuple composed of a function node and the primitives, i.e., operator or terminal (left child, current node, right child), that represent each of its children. Individuals are considered similar if their number of relations surpasses a predefined threshold, zthresh.

In the model, it was implemented as follows:

- For each local optimum in the local optima, it's similarity was compared to with the current potential candidate to see if they were similar. This was done by sending the current potential candiate with each local optimum to the lsim function.

- The function took three arguments: first_individual and second_individual, which are both Node objects, and lthresh, which is an integer threshold.

The function calculated the "local similarity" between the two individuals by computing the intersection of the relation tuples for each individual. If the size of the intersection was greater than lthresh, the function returned true to indicate that the two individuals were similar, otherwise it returned false.

7 Population Control Model

The population control model utilized was a steady-state control model (which assesses if the two children are fitter than certain programs in the parent pool after applying crossover and mutation operators to two selected programs). If they are, they replace the weakest programs in the pool). It was utilized because it lowered the amount of variety loss and improved the likelihood of retaining the best individuals in the population, resulting in faster convergence towards the optimal solution. It is also more memory efficient because it just keeps a certain number of programs in the parent pool.

8 SBGP Approach Employed

The SBGP approach employed was the Iterative Structure-Based Algorithm. The ISBA uses similarity indexes to keep the GP algorithm from exploring locations that are structurally similar to those that have already been visited. This approach can help to overcome the problem of fitness bias that can occur in traditional genetic programming approaches.

Fitness bias is a prevalent issue in genetic programming, in which the algorithm converges prematurely to a suboptimal solution due to the way fitness is evaluated. Fitness is often evaluated using a single objective function in classical genetic programming, which can lead to a bias toward solutions that perform well on that specific target but may not be optimal for the overall problem.

ISBA overcomes this issue by employing a hierarchical representation of the problem domain that captures the problem's underlying structure. This model enables the algorithm to explore the search space in a more directed manner, focusing on the most promising locations. The method analyzes the fitness of the solutions at each level of the hierarchy using numerous objective functions that represent distinct aspects of the problem.

The algorithm has two stages: global search and local search. In the global search stage, the algorithm explores multiple areas of the solution space to find the global optima. In the local search stage, the algorithm performs a more focused search around the best solution found in the global search stage.

Here's a description of the algorithm:

- Searching is done at two levels: global and local.
- At the local level, the process of generating and evaluating individuals solutions is repeated n times.
- Each iteration begins at a fixed starting point (root) determined by the current local optimum found in the global-level search.
- If a solution is found during execution, the search will stop and the solution will be reported.
- If no solution is found, the best individuals are saved as local optima for the local search, and the next iteration is started with a new population that excludes individuals similar to the saved local optima. Similarity is defined as the presence of a fixed component that matches the local optimum stored up to a certain depth (d) at an acceptance threshold (lthresh).
- If no similarity check is performed, the search is repeated several times to avoid local optimization due to random variation.
- If no solution is found after n runs, the search is run m times at the global level.
- At the global level, a set of local optima is maintained, excluding previously visited regions from the search.
- Similarity is defined as a component matching a certain depth (d) using a global acceptance threshold (gsim). For each local optimum found at the global level, n local searches are performed before discarding the entire structure.

Here's pseudocode of the SBGP using ISBA:

```
$m \leftarrow 0$
while $m < max\_number\_of\_global\_runs$ do
  perform run (global run). If this is not the first global run, make sure it
  does not visit the same regions as past global runs. This is accomplished
  by ensuring that each freshly formed individual, selected individual, and
  offspring produced by mutation and crossover differ from the local optima
  representing the areas investigated in prior global runs. This is accomplished
  through the usage of the gsim similarity index.

  if  the individual returned from the run is not the solution then
    Record the individual as a local optimum for global areas

    Identify the fixed component of the local optimum, consisting of the
    subtree comprising the nodes from the root to depth d, that will form
    the first d levels of each element of the all the populations of the local
    level search.

    $n \leftarrow 0$
    while $n < max\_number\_of\_local\_runs$ do
      Perform a run. The elements of all the populations are fixed from the
      root to level d to be the same as the first d levels of the local optimum
      representing the current global area. If this is not the first local area
      visited and similarity checking is required, make sure that this run does
      not visit the same places as earlier local runs. This is accomplished by
      ensuring that each freshly formed individual, selected individual, and
      offspring produced by mutation and crossover are not identical to the
      local optima reflecting the areas explored on earlier local runs. The
      lsim similarity index is employed for this purpose.

      if he individual returned from the run is not the solution then
        Record the individual as a local optimum for local areas
        $n \leftarrow n + 1$
      else
        return the solution
      end if
    end while
    $m \leftarrow m + 1$
  else
    return the solution
  end if
end while
```

9 Canonical Genetic Programming Results

9.1 Root Mean Squared Error

Seed	Train RMSE	Test RMSE	Execution Time (s)
7	17.9457	17.8981	818.3219
21	17.5025	18.4264	769.2596
99	16.7540	16.7209	638.8906
100	17.4778	16.9419	535.1845
101	20.1735	20.2914	614.5707
999	16.7576	16.8565	901.3327
13408	15.1475	15.0396	807.5561
15897	17.2498	16.8647	705.0325
16753	17.3273	17.1572	595.1291
17642	17.8376	17.9687	677.4475
21868	16.4735	16.1346	910.0160

Table 1: RMSE values for different runs with different seeds.

9.2 Median Absolute Error

Seed	Train MedAE	Test MedAE	Execution Time (s)
7	3.8772	3.8902	1468.4388
21	3.3534	3.3528	629.4650
99	5.1876	5.2119	445.1842
100	3.9196	4.0196	554.4415
101	5.2784	5.1371	462.6827
999	4.8695	4.7364	830.1600
13408	5.2975	5.3496	412.2947
15897	5.2964	5.2631	398.8492
16753	5.7460	5.6984	485.5432
17642	6.4061	6.4991	363.2356
21868	5.2902	5.3063	1294.7889

Table 2: MedAE values for different runs with different seeds.

9.3 Mean Absolute Error

Seed	Train MAE	Test MAE	Execution Time (s)
7	11.9467	12.1605	561.4685
21	12.7704	12.8782	437.5668
99	12.4152	12.7110	997.9245
100	9.7307	9.4870	554.4415
101	12.5833	12.2895	542.2013
999	11.5914	11.7246	440.1745
13408	9.5822	9.5798	730.8558
15897	12.8409	12.5922	355.2651
16753	10.9822	10.9686	456.5381
17642	10.2844	10.2716	379.7007
21868	14.4368	14.2321	303.2034

Table 3: MAE values for different runs with different seeds.

Fitness	Avg. Train	Avg. Test	Best Train	Best Test	Avg Execution Time
RMSE	19.0647	19.0299	15.1475	15.0396	797.2741
MedAE	5.4522	5.4511	3.3534	3.3528	734.5084
MAE	13.1164	12.8895	9.7307	9.4870	575.9314

Table 4: Summary of Table 1, Table 2, and Table 3 showing the average and best values for the different fitness functions

10 Structure-Based Genetic Programming Results

10.1 Root Mean Squared Error

Seed	Train RMSE	Test RMSE	Execution Time (s)
14986112	15.3946	15.4066	13058.9986
16223899	19.0404	19.0250	17918.4588
22878896	15.4032	14.4154	16694.1699
68625033	17.0935	17.2016	17735.5211
101006424	15.3303	15.0941	16595.9340
113139835	13.7988	14.0693	16241.2898
125838671	16.0736	15.8404	16074.3270
213232696	16.7318	16.2213	16396.7269
311489368	16.5338	16.5895	15865.3934
376423770	15.7296	15.9865	15991.2868

Table 5: RMSE values for different runs with different seeds.

10.2 Median Absolute Error

Seed	Train MedAE	Test MedAE	Execution Time (s)
14986112	3.7411	3.2803	16319.3152
16223899	3.2912	2.6922	16104.6186
22878896	3.6344	3.1136	16425.9428
68625033	3.9336	2.9712	16131.1718
101006424	3.4734	2.9256	19565.5610
113139835	3.8215	2.9819	18970.8070
125838671	3.6995	3.0231	15970.0389
213232696	3.4697	2.9310	15922.1560
311489368	3.6293	2.9457	15889.0790
376423770	3.4734	2.9256	15715.0674

Table 6: MedAE values for different runs with different seeds.

10.3 Mean Absolute Error

Seed	Train MAE	Test MAE	Execution Time (s)
14986112	9.4964	9.5003	12287.5062
16223899	10.0331	9.1424	15929.3822
22878896	10.1610	9.2841	16024.5888
68625033	10.8271	9.4227	16130.4344
101006424	11.5636	11.0307	15987.3035
113139835	11.4467	10.5046	16477.8029
125838671	10.0069	9.1829	16564.3766
213232696	10.7577	9.3374	16073.1358
311489368	11.7299	10.9978	15935.4528
376423770	14.5369	13.4870	17553.3117

Table 7: MAE values for different runs with different seeds.

Fitness	Avg. Train	Avg. Test	Best Train	Best Test	Avg Execution Time (s)
RMSE	16.1130	15.9850	13.7988	14.0693	16257.2106
MedAE	3.6167	2.9790	3.2912	2.6922	16701.3758
MAE	11.0559	10.1890	10.0331	9.1424	15896.3295

Table 8: Summary of Table 5, Table 6, and Table 7 showing the average and best values for the different fitness functions

11 Discussion Of Results

The model's performance may be described as moderate since the predicted values are relatively close to the actual values, but there is still potential for improvement. Yet, the low values for MedAE suggest that the projected values are close to the actual values, demonstrating that the genetic programming method can accurately forecast bike trip duration.

Overall, the data demonstrated that genetic programming can be a useful method for anticipating bike trip duration. Experimenting with different input features, hyper-parameters, and genetic operators can improve the model's performance even further.

12 Comparison with Performance of Canonical GP Implemented in Assignment 1

Here is how the model compared to the canonical GP implemented in assignment 1 using the best results it achieved for the various fitness functions:

- Root Mean Squared Error (RMSE): The model outperformed the canonical GP model.
- Median Absolute Error (MedAE): The model outperformed the canonical GP model
- Mean Absolute Error (MAE): TThe model outperformed the canonical GP model

13 Comparison with Performance of Approaches in Research Paper

Here is how the model compared to the other models in the study report using the best results it achieved for the various fitness functions:

- Root Mean Squared Error (RMSE): The model outperformed the LR model but did not outperform the GBM, KNN, and RF models.
- Median Absolute Error (MedAE): The model outperformed the LR and GBM models but did not outperform the KNN, or RF models.
- Mean Absolute Error (MAE): The model outperformed the LR model but did not outperform the GBM, KNN, or RF models.

14 Conclusion

The Median Absolute Error was the best fitness function after the model was trained and evaluated using a variety of fitness functions, earning a score of 2.6922. While its performance with the KNN model was close, it outperformed 2 of the models in the study article. It fared better compared to the canonical GP. This demonstrated the value of structure-based genetic programming in estimating the duration of a bicycle trip.

15 Extra Notes

- A subset of the data (100 000 samples) were used because of the following:
  - Lack of computing power - Because my machine is an i5, it took a long time to execute the model. I couldn't leave my laptop running for hours since I needed it to do my work and other personal duties. Google Colab's free resources were not always available.
  - When I utilized the entire dataset vs a portion of the data, the findings were same. The only difference was the execution timings, which were around 6 hours for a single run, which was inefficient.
  - In the config.py file, you can change the sample size to be greater or smaller than 100 000.
- I performed the following to reduce the execution times:
  - The function that evaluated the tree was converted into fast machine code (through Just-In-Time compilation) using the numba package to minimize the time it took to determine the fitness of a each program in population of size 50 with many runs and generations.
  - I attempted to use a free GPU from Google Colab, but I was restricted in time (30 minutes). None of the other cloud computing platforms provided free GPUs.
  - Recursion was avoided and iteration was used to build some functions that operated on a program (tree).
- Please refer to the README for instructions on how to run the model.