COS 710: Assignment III

Structure-Based Genetic Programming

u22498037
University of Pretoria

Contents

1 Running Instructions

To run the GP, made with C++, a `makefile` is included as part of the submission. The following commands in the directory can be used to run the GP:

```
make
make run
```

Since architectures may vary, the following can be commands can also be used to (hopefully) run the GP when `clang` is installed:

```
make alt
make run
```

Running this on a Macbook Air M1 with 16GB RAM averaged 84.53 seconds per structure-based GP run and 54.59 seconds per normal GP run. There is no need to run the preprocessing script again.

2 Notation

The following abbreviations is used for the remainder of this report:

1. **SBGP:** Structure-based Genetic Programming
2. **NGP:** Normal Genetic Programming

3 Exploratory Data Analysis (EDA)

To understand the data, Exploratory Data Analysis (EDA) were done to understand the relationship between the different datasets to discover any inconsistencies. In Section 4, the dataset will be processed based on the EDA.

It was evident that the target variable is imbalanced (Figure 1, bottom right) where `1.0` is live and `0.0` is die. This impacted the fitness function described in Section 6.2. Other variables such as sex, antivirals, and anorexia can also considered to be skewed (Figure 1).

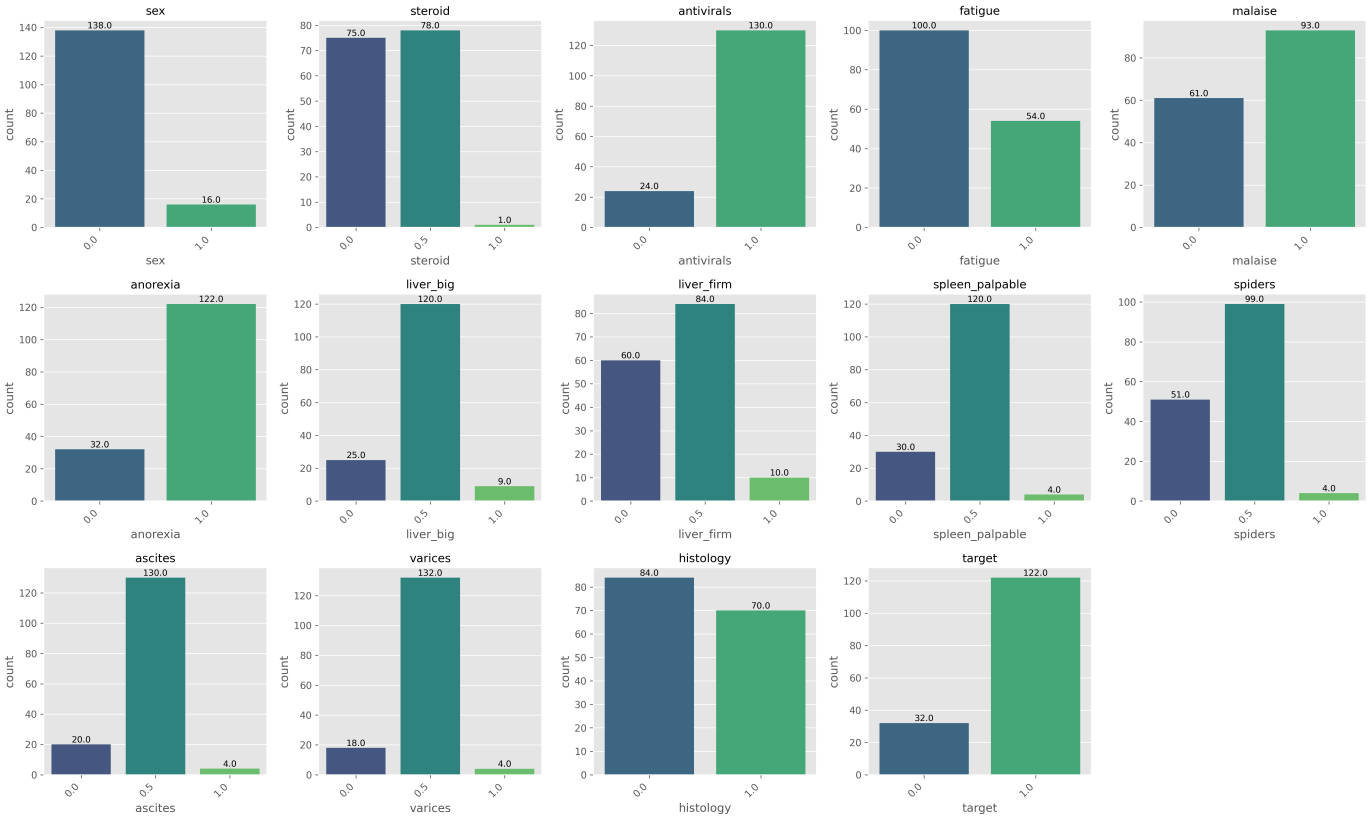

Figure 1: A histogram visualising the distribution of all the features.

4 Preprocessing

4.1 Duplicates

No duplicates were detected and therefore no further action were taken in this regard.

4.2 Outliers

To remove the outlier entries, data points with a ± 3 Z-score (an indication of how many standard deviations the point is away from the mean) were removed. Removing outliers helped the model to converge better due to more consistent tree outputs. Since only one row were removed, it didn't greatly impact the model's ability to convergence, but the outlier was removed regardless of the outcome.

4.3 Skewness

As noted in Section 3, a few of the variables, including the target variable, were skewed. No further preprocessing was done with regards to skewness as the fitness function (described in Section 6.2) catered for the imbalanced data.

Experiments were conducted by upsampling the target variable to balance the dataset, but it caused the rest of the features to lose meaning when training.

4.4 Column-wise Min-Max Normalisation

The data was normalised column-wise using min-max scaling [1]:

$$X' = \frac{X - min}{max - min} \tag{1}$$

This helped to preserve each column's features and its respective scale. Although *tanh*, the fitness function, resides in the range $(-1, 1)$, it was still essential to scale the dataset to be on the same scale. This brought the continuous variables in range with the discrete (boolean) values, allowing to compare these with comparison operators.

4.5 Feature Scaling

Some features, especially the binary variables such as the target variable, was initially put as `1.0` and `2.0`. Normalisation helped to scale the values between `0.0` and `1.0` regardless if it's a binary variable or not.

4.6 Correlation

After performing preprocessing, the correlation plots was used to understand how these variables interact with the target variable:

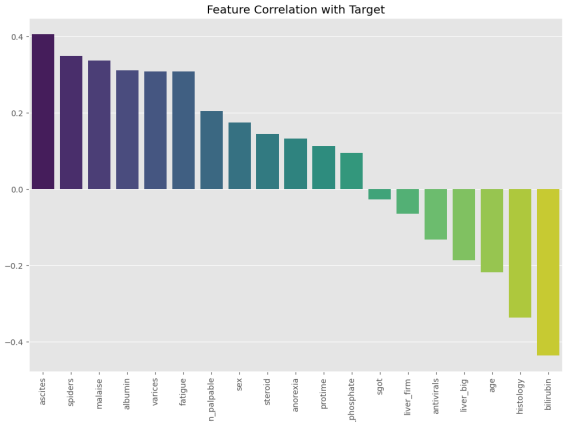

Figure 2: Correlation plot of all the variables with regards to the target variable.

As can be seen, the correlation plots exhibit a well-balanced feature selection where some features negatively correlate with the target variable, whereas others positively correlate to the target variable. Due to the stochastic nature of Genetic Programming, this helped with convergence.

5 GP Runtime Performance

To improve the performance of the GP during runtime, C++'s pass by reference ability was utilised throughout the codebase to further improve performance. Because of the small dataset, no other optimisations have been done.

6 GP Technical Specification

6.1 GP Setup

Both the SBGP and the NGP followed the same setup apart from small alterations in some parts of the setup when needed for the SBGP. This allowed to work from the same GP implementation, reducing the overall code footprint.

6.1.1 Control Model

To control the population of both the NGP and SBGP, a **steady-state control model (SSCM)** has been employed:

---

1. Generate initial population
2. Repeat *until termination criteria is met*
   2.1. Tournament selection (winners)
   2.2. Inverse tournament selection (losers)
   2.3. Generate offspring by applying genetic operators to winners
   2.4. Replace losers with offspring

With SSCM, a fixed population size is maintained throughout the run where the offspring replaces individuals with poor fitness. The method to select the worst individuals is described in Section 6.1.7. By replacing the worst individuals, top-performing individuals are preserved. This helps with steady convergence. On the other hand, a generational control model could potentially disrupt those individuals if no mechanism is put in place.

6.1.2 Structure-based GP (SBGP) Specification

To implement SBGP, a similar approach has been followed to ISBGP-II, as proposed by Kapoor and Pillay [2]. This specific structure-based approach employs global and local similarity indices to adjust the search strategy. Global search allows for wider exploration, while the indication of a promising individual further exploits the program. Once local search fails it returns to global search. This essentially escapes local optima. The SSCM model (Section 6.1.1) is modified for SBGP as follows:

---

1. Generate initial population
2. **Enable global search**
3. Repeat *until termination criteria is met*
   - 3.1. Tournament selection (winners)
   - 3.2. Inverse tournament selection (losers)
   - 3.3. Generate offspring by applying genetic operators to winners
   - 3.4. **If global search is enabled**
     - 3.4.1. **Generate global index for offspring**
     - 3.4.2. **If index is below `globalThreshold`, replace losers with offspring and disable global search.** (we found a promising individual)
   - 3.5. **If global search is disabled** (i.e. local search)
     - 3.5.1. **Generate local index for offspring**
     - 3.5.2. **If index is below `localThreshold`, replace losers with offspring.**
     - 3.5.3. **Otherwise enable global search** (we possibly reached a local optima)

6.1.3 Representation

The individuals in the population were represented as a tree. Each subtree can have up to two subtrees, where `children` denotes one or more subtrees. The functions (non-terminal nodes) were mathematical and/or logical operators and the terminals (terminal nodes) were features or real numbers. The output of the tree is first passed into a `tanh`-function before being converted to a binary number by means of a predefined threshold. If the tree happen to be invalid, it will bypass the fitness calculation (described in Section 6.2) and output the worst possible fitness, which is `0.0` in this case (0% accuracy).

6.1.4 Function and Terminal Set

`randomTerminal()` and `randomOperator()` functions determined which terminal or operator is chosen when generating a tree. Arithmetic operators (`+,-,/,*`) helps to combine continuous features. Non-linear functions (`tanh,sin,cos,log`) allows to model non-linear interactions and helps with scaling values more or less between -2.0 and 2.0. Conditional operators also helped with scaling (e.g. two values compared outputs either 0.0 or 1.0) but also helped with *routing* the output, such as when a subtree becomes irrelevant. In terms of the terminal set, all variables were kept as they all influenced the target variable (Figure 2). `double` were also added, which helps with bias offset and scaling. The GP's function and terminal set:

```
F = {+, -, /, *, max, min, tanh, sin, cos, log,
    <, >, <=, >=, ==, !=}
T = {double, age, sex, steroid, antivirals, fatigue,
  malaise, anorexia, liver_big, liver_firm,
  spleen_palpable, spiders, ascites, varices,
  bilirubin, alk_phosphate, sgot, albumin,
  protime, histology}
```

where `double` is a random real number between -0.5 and 0.5. The features from the dataset differed between continuous and discrete variables, but could co-occur. Experimentation were done to separate continuous and discrete features, but it turned out that if the trees were discrete close to the root, it barely used continuous features elsewhere, causing the tree to be biased towards one class. Since the continuous variables are normalised between 0.0 and 1.0, it worked well with the discrete variables either 0.0 or 1.0.

6.1.5 Train/test split

A standard 80%/20% train split was used. The datasets were shuffled before splitting them for training and testing to ensure randomness. This training split gave enough data to fit without leaving too little data for testing.

6.1.6 Initial Population

The initial population was generated using a randomised grow method with `maxDepth` as a generation-specific parameter. For each iteration, two booleans would independently determine if a left or right subtree would be further generated or not. The chance of these two booleans being true is increasingly determined by the current `maxDepth`. In the recursive function, `maxDepth` decreases as the tree grows. This generation method helped to start off with a diverse population of various depths.

6.1.7 Tournament Selection

For the selection method, a tried-and-trusted tournament selection has been used. This selection method selects n random candidates to potentially apply genetic operators to. The two candidates with the best fitness is selected as parents. If the mutation operator is applied, the best parent is selected. Intuitively, for crossover, both parents are passed as parameters, and for reproduction, the two parents are passed onto the next generation.

To select individuals with poor fitness, **inverse tournament selection** has also been employed to find the two worst candidates to be replaced with the offspring.

Tournament selection was chosen to offer a balance between exploitation and exploration. It's also computationally efficient, since only the fitness of a few individuals in the population have to be calculated.

6.1.8 Mutation

For the mutation genetic operator, a mix of approaches has been followed to ensure maximum diversity. A random point in the tree is chosen, and a random subtree (possibly just a leaf node) is generated and the subtree is replaced. This ensures that both grow and shrink mutation is possible with this approach. The mutation operator also employed point mutation (50% chance), which replaced the node with a terminal or a function, depending on it's position.

The root node is also replacable, but because `rand() mod maxDepth` is passed as parameter, it becomes possible to penalise bigger trees, by generating a new smaller tree (i.e. the root is replaced by a smaller tree). Having a replacable root node is potentially harmful to the individual, but no notable problems have been picked up.

For SBGP, if the mutation point was above the cutoff depth during local search, mutation was skipped.

The combination of subtree and point mutation offers multiple ways to explore new regions. Mutation is a good genetic operator to bring in completely new elements to the program space.

6.1.9 Crossover

The crossover operator takes a standard approach by selecting random points in each of the two parents, and then swaps them around to produce offspring. The root nodes can't be swapped, because pointers would then be swopped, which caused upstream problems when replacing offspring with the losers.

For SBGP, if any of the crossover points were above the cutoff depth during local search, the crossover attempt was cancelled. However, a retry mechanism was added to attempt other points in the tree to apply crossover to.

Crossover offers the opportunity to swop partially good solutions with each other in the program space. Moreover, the offspring are generally fit as their parents are fit as well. Crossover is a good choice for protecting good solutions, and seeking a solution that is potentially better.

6.1.10 Reproduction

The selected individuals remains unchanged and is carried over to the next generation. In the codebase, the reproduction rate is calculated as $1.0 - C_r - M_r$ where C_r is crossover rate and M_r is mutation rate. Reproduction helps to preserve fit individuals.

6.1.11 Stopping Criteria

For the stopping criterion, every GP run completed the full amount of generations (i.e. `maxGenerations`). No experimentation with early stopping or convergence testing has been done in this assignment. The final result were used to understand where the best convergence point were for multiple generations. Experiments with different `maxGenerations` has been done to see which one performs best.

Fixing the number of generations ensures that runs can converge sufficiently, whereas early stopping could potentially accept an average solution over a better solution.

6.2 Fitness Function: BACC

To evaluate the performance of the GP, Balanced Accuracy (BACC) has been used as the `fitness()`-function. BACC works well for datasets that are imbalanced by making sure that both positive and negative classes are weighted equally. Sensitivity accounts for the positive classes whereas specificity caters for the negative classes. The average between these two are taken and is be defined as follows:

$$BACC = \frac{\text{specificity} + \text{sensitivity}}{2.0} \tag{2}$$

where specificity $= \frac{TN}{TN+FP}$ and sensitivity $= \frac{TP}{TP+FN}$. If $TP + FN = 0.0$ or $TN + FP = 0.0$, a bad fitness of 0.0 was returned.

To ensure the threshold can be applied when calculating BACC, the *tanh* function was used to ensure the output resides around 0.0:

$$f(x) = tanh(x) \tag{3}$$

The *tanh* function ensures smooth, bounded outputs between -1.0 and 1.0, which improves numeric stability (no overflow/underflow).

Because the *tanh* function outputs continuous values, a clipping mechamism was used:

$$\text{clip}(x) = \begin{cases} 1.0 & \text{if } x > \text{threshold} \\ 0.0 & \text{otherwise} \end{cases} \tag{4}$$

where the threshold were set to 0.0. If the tree output was 0.0 (i.e. die) then $tanh(0) = 0$. Before starting with optimising parameters, F1-score and normal accuracy have also been experimented with, as well as an upsampled dataset, but BACC with a normal dataset was chosen to balance the predictions.

7 Parameters and Results

The parameters were manually tuned to achieve optimal performance. A wide range of variables have been experimented with. Reasoning have been provided next to each parameter.

7.1 NGP Parameters

After fine-tuning the parameters, the following NGP parameter setup is used:

- Population size: **35** *(Greater population size led to increased runtime with minimal performance improvements.)*
- Number of generations: **70** *(A greater number of generations led to increased runtime. The GP already converged and didn't notably impact the results.)*
- Crossover rate: **0.6** *(Higher crossover rates reduced convergence. A reasonably low crossover rate helped for convergence.)*
- Mutation rate: **0.25** *(A high mutation rate was selected to allow the model to naturally escape local optima.)*
- Reproduction rate: **0.15** *(The reproduction rate was simply a fallback and simply transfers the parents to the next generation.)*
- Max Depth: **7** *(The initial depth is set to cater for the amount of variables and allow for genetic diversity. Furthermore, this allowed trees to have a wider range of depths due to the initial population generation method described in Section 6.1.6.)*
- Tournament Size: **7** *(The tournament size was made quite big to explore a big search space (20% of the population size).)*
- Run n's seed i is $i = n - 1$. *(This remained unchanged from the start.)*

7.2 SBGP Parameters

After fine-tuning the parameters, the following SBGP parameter setup is used:

- Population size: **35** *(Greater population size led to increased runtime with minimal performance improvements.)*
- Number of generations: **110** *(More generations were required than the NGP, likely due to the application of the local and global threshold.)*
- Crossover rate: **0.5** *(Crossover did not seem to help too much with the convergence, and was therefore kept on the low side. This value is mostly inspired by Kapoor and Pillay [2].)*

- Mutation rate: **0.25** *(The mutation rate were set high to escape possible local optima and to allow solutions to be accepted. This allowed for genetic diversity throughout the run. This value is also inspired by Kapoor and Pillay [2].)*
- Reproduction rate: **0.25** *(The reproduction rate was simply a fallback and simply transfers the parents to the next generation.)*
- Max Depth: **6** *(A smaller number than NGP's max depth was chosen to cater for the thresholding mechanism. There's a strong correlation between this parameter and the cutoff depth. This is not the strict max depth, but rather the initial max depth.)*
- Tournament Size: **7** *(The tournament size was made quite big to explore a big search space (20% of the population size), which helped for exploration.)*
- Run n's seed i is $i = n - 1$. *(This remained unchanged from the start.)*

On top of that, specific SBGP parameters were added:

- Local threshold: **8** *(Manual tuning were done to find the best value for overall performance.)*
- Global threshold: **6** *(Manual tuning were done to find the best value for overall performance.)*
- Cutoff Depth: **4** *(Manual tuning were done to find the best value for overall performance. As mentioned before, this value is correlated with the initial maximum depth.)*

7.3 Results

The following section will exhibit the results, but will only be discussed in Section 8. Captions were added to explain the figures and tables.

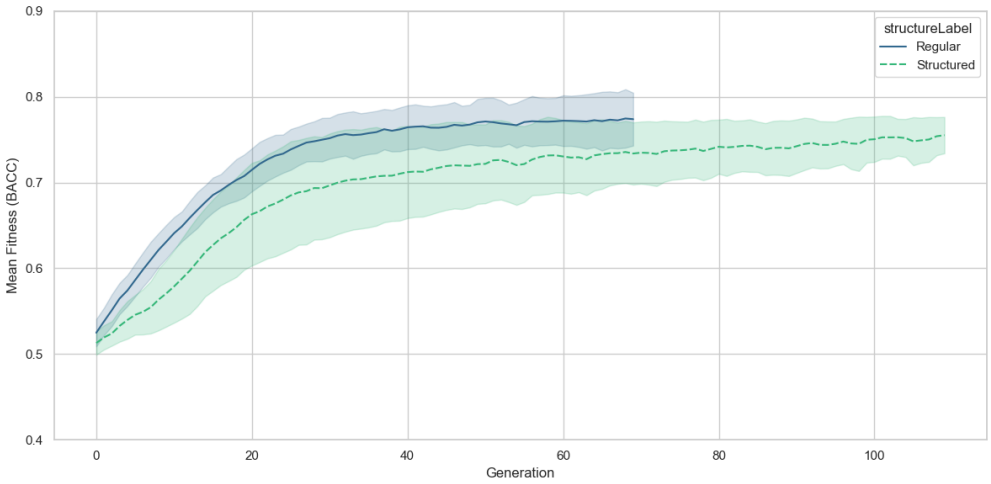

Figure 3: Population fitness over multiple runs. It's evident in this figure that SBGP required more runs to converge whereas NGP runs converged sufficiently after 75 generations. NGP's runs were more stable, where SBGP likely depended more on whether to escape local optima.

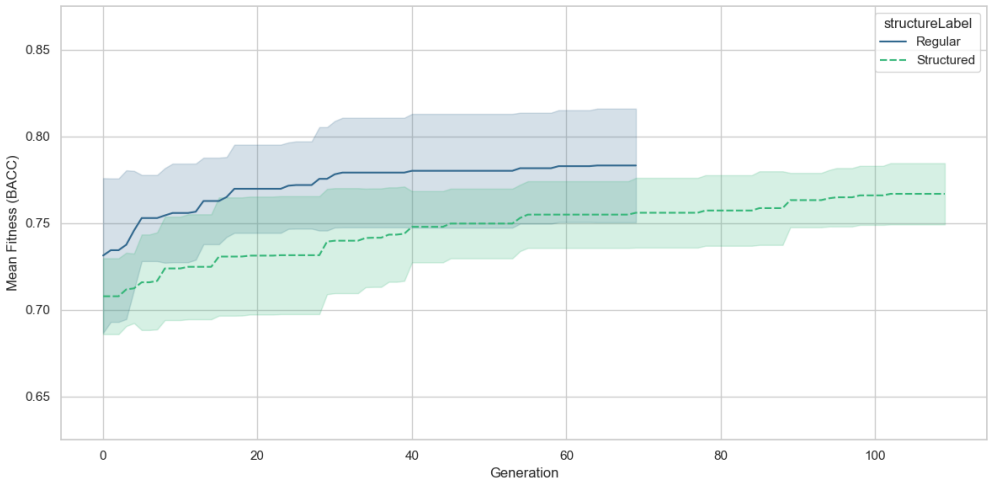

Figure 4: Progression of best tree fitness per generation for both the NGP and SBGP variants across all runs.

7.3.1 Training Results

	Best Tree Statistics (BACC Fitness)				*Population Statistics (BACC Fitness)*			
Run	Mean	Min	Max	Std	Mean	Min	Max	Std
0	0.803983	0.778367	0.822653	0.016623	0.739508	0.514927	0.814245	0.082290
1	0.751166	0.669184	0.772449	0.025206	0.713052	0.529096	0.762163	0.064955
2	0.796879	0.783460	0.798611	0.004855	0.743261	0.504960	0.798611	0.079888
3	0.805149	0.757137	0.823751	0.023970	0.739562	0.542800	0.813062	0.072494
4	0.738013	0.674060	0.743421	0.013222	0.690625	0.503136	0.743330	0.070118
5	0.768891	0.747685	0.773727	0.010200	0.729921	0.539964	0.773727	0.060717
6	0.784124	0.756970	0.795304	0.007241	0.732184	0.530353	0.784514	0.071208
7	0.774320	0.674060	0.799248	0.035074	0.717953	0.520440	0.797385	0.077072
8	0.776458	0.757143	0.782653	0.011018	0.728679	0.514070	0.782653	0.072485
9	0.719593	0.714512	0.719667	0.000616	0.692540	0.549054	0.719667	0.038272
Overall	**0.771858**	0.669184	0.823751	0.028398	0.722728	0.503136	0.814245	0.018919

Table 1: **NGP Training Results:** Best Tree Fitness and Population Fitness statistics per GP run. The Overall row shows the mean across runs, the best result achieved in any run, the worst best result in any run, and the standard deviation of means across runs.

	Best Tree Statistics (BACC Fitness)				*Population Statistics (BACC Fitness)*			
Run	Mean	Min	Max	Std	Mean	Min	Max	Std
0	0.733725	0.690000	0.782245	0.035523	0.631417	0.485516	0.744023	0.082394
1	0.730408	0.730408	0.730408	0.000000	0.692305	0.527819	0.730408	0.056291
2	0.726601	0.717803	0.763889	0.018196	0.639497	0.515711	0.739412	0.057466
3	0.752511	0.686360	0.774980	0.022280	0.712234	0.525382	0.774980	0.066870
4	0.740138	0.676128	0.761090	0.035194	0.703818	0.508002	0.761090	0.070182
5	0.761458	0.690972	0.782407	0.021149	0.723699	0.511541	0.776141	0.067833
6	0.741031	0.725055	0.742480	0.004538	0.705974	0.516754	0.742480	0.053186
7	0.740432	0.701316	0.777068	0.029074	0.709218	0.528958	0.774060	0.059437
8	0.775965	0.742653	0.777551	0.007302	0.747951	0.510391	0.777551	0.063148
9	0.767915	0.716891	0.775971	0.020368	0.730369	0.498425	0.775971	0.076455
Overall	**0.747018**	0.676128	0.782407	0.016697	0.699648	0.485516	0.777551	0.037253

Table 2: **SBGP Training Results:** Best Tree Fitness and Population Fitness statistics per SBGP run. The Overall row shows the mean across runs, the best result achieved in any run, the worst best result in any run, and the standard deviation of means across runs.

7.3.2 Testing Results

Run	NGP	SBGP	Difference
1	0.711310	**0.782738**	+0.071428
2	**0.803571**	0.729167	−0.074404
3	0.660326	**0.785326**	+0.125000
4	0.673333	**0.796667**	+0.123334
5	**0.907407**	0.708333	−0.199074
6	0.761538	**0.884615**	+0.123077
7	0.750000	**0.892857**	+0.142857
8	0.638889	**0.833333**	+0.194444
9	0.732143	**0.782738**	+0.050595
10	0.590000	**0.656667**	+0.066667
Average	0.722852	**0.785244**	+0.062392

Table 3: **Test Results:** Comparison of NGP and SBGP fitness (BACC) across 10 runs. Positive differences indicate better performance by SBGP.

8 Discussion

SBGP is an interesting avenue to build and train a GP model for data classification. Different than a normal GP implementation, SBGP attempts to control the search space, but comes with a number of challenges. During experimentation, as seen in Figure 3 and Figure 4, it is noted that the SBGP required more generations to converge similarly to the baseline GP (NGP). This is likely due to the *gating* that the local and global threshold employs, where some good-enough solutions might be rejected due to possible local optima.

Even though NGP converged, there were signs of overfitting as the training accuracy were lower than the testing accuracy. However, with the SBGP, underfitting is noted since the mean test accuracy is higher than the mean train accuracy. The case for underfitting could likely be traced back to the dataset being relatively small.

In both cases, the mechanisms employed in the fitness function played an important role to accurately measure the outcomes of the individuals. It's also observed in Table 1 and Table 2 that SBGP's population's standard deviation is almost double NGP's standard deviation, indicating better diversity and exploration.

With the aim of maximising BACC, SBGP outperforms the NGP on testing data by 6.24% (Table 3). However, the SBGP struggled when the parameters were not tuned well, and it's likely that there are better parameters for this specific solution. Further work could include a more fine-grained approach to determining exact function and terminal sets. Due to time constraints, no automated parameter fine-tuning methods were explored, which could potentially improve the overall solution. Another addition could be to threshold continuous

variables to act as boolean variables, but is not guaranteed to work. Other genetic operators such as the edit operator could also help to identify individuals with local optima and replace nodes in the individual tree with better nodes. This is not necessarily meant to directly improve fitness, but rather to ensure genetic diversity and mitigate local optima. Data scarcity was a problem in this assignment, and preprocessing techniques such as upsampling can be explored to potentially expose better performance.

In conclusion, SBGP is not domain-specific and can readily be adapted to other domains and datasets with sufficient parameter fine-tuning and GP setup. It is shown in this report that by controlling the search space, performance increases can be observed.

References

[1] S. G. K. Patro and K. K. Sahu, "Normalization: A preprocessing stage," *arXiv:1503.06462 [cs]*, 03 2015. [Online]. Available: https://arxiv.org/abs/1503.06462

[2] R. Kapoor and N. Pillay, "A genetic programming approach to the automated design of CNN models for image classification and video shorts creation," *Genetic Programming and Evolvable Machines*, vol. 25, no. 1, p. 10, Mar. 2024. [Online]. Available: https://doi.org/10.1007/s10710-024-09483-5