COS 710:A2 Report

Kaleb Bruwer

May 7, 2022

1 Data pre-processing

Before starting, I did some analysis on the data to decide on input values and types. Firstly, a few records had to be removed from the end of the Cleveland dataset since they were corrupted. A few fields have been excluded for having zero variance, meaning they carried zero information. Furthermore I divided the data up into floating point and integer fields, where every field that has a floating point value for at least one record in the dataset is always treated as a floating point. This left me with 8 floating point and 61 integer inputs, giving a total of 69 inputs. The target is also an integer.

The target is an integer number between (and including) 0 to 4. This represents the level of heart disease that a patient has. The goal is to predict this number based on the other fields in the dataset. Not all of the fields in the dataset are always populated and missing values are represented by -9.

2 The GP algorithm

2.1 Representation

2.1.1 Structure

An arithmetic tree was used, although some algorithmic/logical functions are supported (but not used in the final parameters). Each genetic program is a tree, although in the implementation the tree is unpacked into a 1D array. This makes the fitness evaluation very efficient since it can be done by simply iterating through the array and keeping intermediate values on a stack. There are four kinds of nodes in the tree: functions, consts, int inputs, and float inputs. The different input types are separated for the sake of the type system, since the GP is generated and altered in accordance with a strict type system. All consts are of type int and have values in the range [-32,31].

2.1.2 Functions

Functions are further subdivided by return type, which is either float or int. A few boolean functions also exist, but they are treated as int functions. The following is the list of functions:

Return type	Function	1st input	2nd input	3rd input
INT	+	INT	INT	
INT	-	INT	INT	
INT	*	INT	INT	
INT	/	INT	INT	
INT	mod	INT	INT	
INT	round	FLOAT		
INT	IF	INT	INT	INT
INT	==	INT	INT	
INT	==	FLOAT	FLOAT	
FLOAT	+	FLOAT	FLOAT	
FLOAT	-	FLOAT	FLOAT	
FLOAT	*	FLOAT	FLOAT	
FLOAT	/	FLOAT	FLOAT	
FLOAT	toFloat	INT		
FLOAT	IF	INT	FLOAT	FLOAT

Due to the strict type system, sever functions have both INT and FLOAT versions. However, there are round and toFloat functions to convert between the two types. These are given higher weights (see the parameters) to prevent the tree from getting "stuck" in one subset of input values.

2.2 Fitness function

The miss rate is used as the fitness function, i.e. the fraction of training cases for which the result was wrong. In addition to this, a regularization term is used to manage the tree size. The number of nodes in a tree, scaled by some factor, is added to the fitness value. A lower fitness is better, which is why the miss rate is used instead of the hit rate. The regularisation factor is one of the parameters.

Mean squared error (MSE) was also tested, but did not perform as well as miss rate. This is likely because miss rate is so strongly correlated to how accuracy is measured, so directly optimizing for the target metric gives the best results. MSE would sometimes match miss rate's performance, but other times it would get stuck in a local minima where the fitness is low, but the testing accuracy is poor.

2.3 Selection

Tournament selection was used. The advantage of this as opposed to just selecting the best individuals over the full population is that it slows down convergence. If the best individuals are selected over the full population, the selection will rapidly become dominated by the best individual, replicating itself exponentially faster with each generation until the entire population is converged on it. With tournament selection, it will take much longer for such a tree to be placed in enough tournaments to replace everything else, meaning other trees will have more time to make a viable mutation.

In my version of tournament selection there are three parameters: the tournament size, the number of winners and the multiplication factor. The tournament size is how many individuals are placed in a tournament. Then, a certain number of winners is picked in the tournament. Finally, losers are picked from the bottom of the tournament so that every winner can be cloned over as many losers as the multiplication factor. So for example, if the settings were size=8, winners=2, multiplication=2, then in a tournament of eight, two winners would be picked, each of which would replace two of the worst performers, meaning 4 individuals would be replaced.

Finally, the clones are put in a pool for genetic operators to be run on. Therefore the winners and ignored individuals are reproduced into the next population, with the clones of the winners (cloned over the losers) being altered by the genetic operators (excl. reproduction).

2.4 Genetic operators

- **Reproduction:** This was the "default" operator, implicitly applied to every individual that wasn't replaced.
- **Mutation:** A random node in the tree is selected, the subtree starting there is deleted and replaced by a new randomly generated subtree.
- **Crossover:** Two individuals are taken and a random subtree (of the same type) is selected in each. These subtrees are then swapped. Since the subtrees selected are taken from effective trees, they are more likely to "mean" something than a randomly generated subtree (as is used in mutation). Simultaneously, there is far less variation possible with crossover, since it is constrained by what is already in the population. It is therefore a convering operator.
- **isEmpty mutation:** (NOT USED, BUT IS SUPPORTED) This constructs a special structure that returns a particular input if it's not empty (!= -9), otherwise it will return the result from a randomly generated subtree, much like a a normal mutation. This was added in an attempt at letting the GP more easily deal with missing inputs, which are represented by -9 in the dataset. This turned out to be counterproductive, so has a weight of 0 (see the parameters section), and is therefore never used.

2.5 Termination criterion

Training terminates after a predetermined number of generations, which is treated as one of the parameters.

3 Parameter values

All of these parameters can be found and tweaked in the Parameters.h file. For changes to take effect, the code has to be recompiled.

```
POPULATION = 10000;
INIT_DEPTH = 5;
GENERATIONS = 200;
TRAIN_COUNT = 700;
```

The population is always generated by the ramped half-half method. A max depth of 5 was chosen, but the trees can grow much bigger through mutation with no hard limit to stop them; it is up to regularization to keep tree-size in control. The first 700 of the 899 values are used as training data, with the last 199 being testing data.

```
TOURNAMENT_SIZE = 6;
TOURNAMENT_WINNERS = 1;
TOURNAMENT_MULTIPLICATION = 3;
```

A tournament size of 6 is used where one winner is picked, which replaces the 3 worst-performers.

```
REGULARIZATION_WEIGHT = 0.0004;
```

As mentioned in the fitness function, this factor determines the tree size's impact on fitness. A higher number means larger trees are more severely penalised. This number may be small, but it has a strong impact since it makes smaller trees win out over bigger ones with similar performance.

```
CONST_CHANCE = 200; //fraction out of 1000
GROW_RATE = 750;
```

These parameters influence how trees are generated. CONST_CHANCE is the likelihood of a terminal node being a constant value instead of an input. GROW_RATE is the likelihood of a function node being added instead of a terminal node when generating trees with the grow method. If it is too low, generated trees will rarely grow beyond a small size. If it is too high, it will be similar to full trees. These numbers are fractions out of 1000, so 200 actually means a likelihood of 0.2.

```
[Crossover, Mutation, isEmpty]
OP_WEIGHTS[] = {7,1,0};

[+, -, *, /, toFloat, IF]
FLOAT_WEIGHTS[] = {1,1,0,1,3,0};

[+, -, *, /, mod, round, IF]
INT_WEIGHTS[] = {1,1,0,1,1,2,0};
```

OP_WEIGHTS stores the relative likelihoods of different genetic operators being chosen. The crossover to mutation balance had to be carefully chosen since it strongly impacts the convergence rate.

The FLOAT and INT WEIGHTS arrays are the relative likelihoods of different functions being picked when creating random function nodes. IF has been removed since experimentation showed it had a tendency to overfit. (Training accuracy went up, but testing accuracy stayed low). Multiplication has also been removed for giving far too large output values.

4 Accuracy

More data can be found in the Results folder. Those files include per-generation data on all 10 runs and also has the fitness. Since the per-generation data for the 10 runs combined totals to 2000 lines, it would be impractical to put in this document.

I measured accuracy as the hit rate, i.e. the fraction of predictions that are correct. Table 1 shows the accuracy for the 10 runs. Table 2 shows the statistical analysis of the columns in Table 1, for example the average best accuracy over the 10 runs.

Run	Training			Testing		
	Best	Average	Std Dev	Best	Average	Std Dev
0	0.89	0.78	0.37	0.80	0.67	0.40
1	0.94	0.81	0.39	0.85	0.71	0.41
2	0.89	0.79	0.33	0.85	0.69	0.44
3	0.91	0.81	0.33	0.85	0.72	0.39
4	0.87	0.77	0.33	0.85	0.69	0.42
5	0.76	0.64	0.39	0.72	0.55	0.43
6	0.87	0.74	0.36	0.85	0.65	0.47
7	0.84	0.71	0.39	0.76	0.58	0.43
8	0.79	0.63	0.41	0.72	0.52	0.44
9	0.97	0.84	0.41	0.87	0.74	0.41

Table 1: Accuracy across all 10 runs

Statistical Attribute	Training			Testing		
	Best	Average	Std Dev	Best	Average	Std Dev
Best	0.97	0.84	0.33	0.87	0.74	0.39
Average	0.87	0.75	0.37	0.81	0.65	0.42
Std Dev	0.064	0.072	0.032	0.058	0.076	0.023

Table 2: Consistency of runs

5 Performance

It should be noted that the training set is vastly different from the testing set. Much of the testing data comes from the "switzerland.data" file which has a very different class distribution from the training data. This is actually ideal, since "out of distribution" data gives a better indicator to whether the GP learned something as opposed to just overfitting.

An algorithmic tree has proven to be a powerful and effective solution for this particular problem, often obtaining training accuracies above 95% and testing accuracies around 85%. Additionally, this was achieved in approximately 2 minutes run on an ordinary 8-thread laptop.