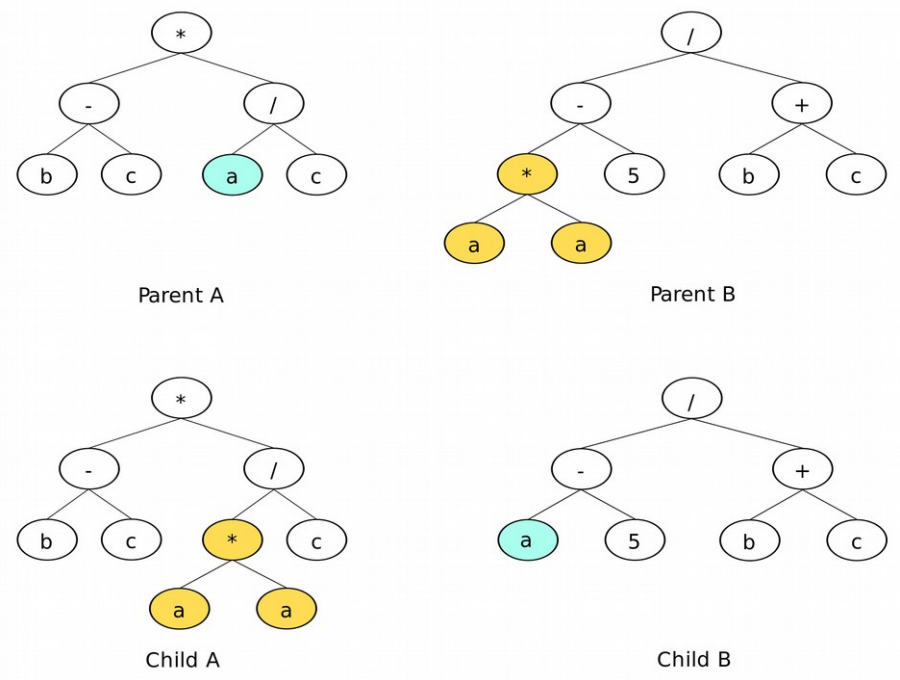
Karoo GP User Guide

Genetic Programming in Python

ver. 20190608

by Kai Staats

kstaats.github.io/karoo_gp/



Karoo GP

User Guide

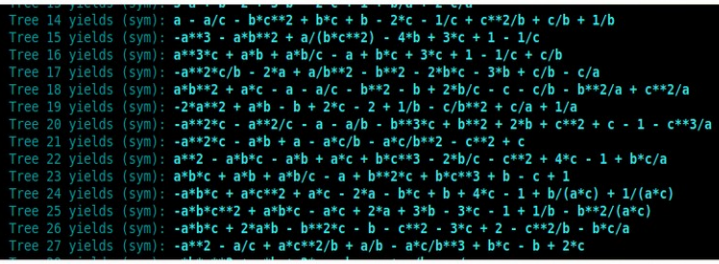
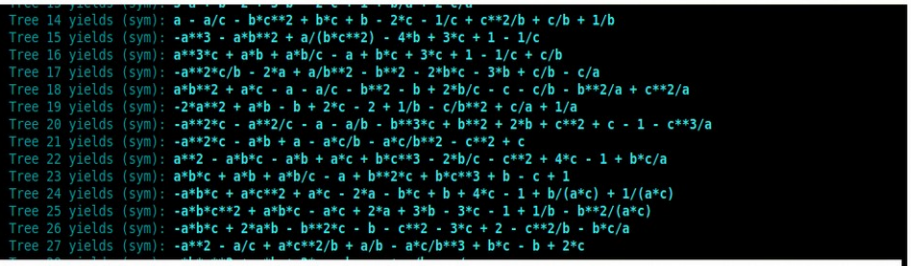


Table of Contents



Welcome to Karoo GP

Karoo GP is a Machine Learning application suite which provides both symbolic regression and classification data analysis. Written in the programming language Python, Karoo GP owes its foundation to the "Field Guide to Genetic Programming" by Poli, Langdon, McPhee, and Koza.

Karoo GP is a scalable platform with multicore and GPU support, designed to readily work with realworld data. *No programming required.* As a teaching tool, it enables instructors to share step-by-step how an evolutionary algorithm arrives to its solution. As a hands-on learning tool, Karoo GP supports rapid, repeatable experimentation. Karoo GP offers an intuitive Desktop user interface and a fully scriptable Server interface with user defined default parameters and command-line arguments, both of which work with your datasets.

This guide is designed to give you a crash course in tree-based Genetic Programming and a light introduction to preparing datasets for machine learning. Even if you are an expert in machine learning, you are encouraged to make time to read this document in order to anticipate all that Karoo GP has to offer.

Requirements:

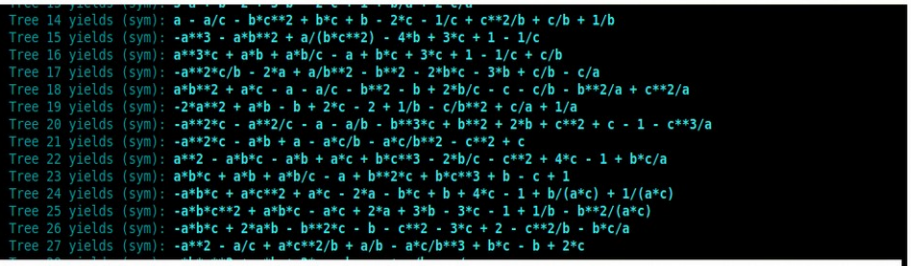
- python 3.6 with numpy 1.16.4, sympy 1.4, tensorflow 1.13.1, scikit-learn 0.21.2

What is Genetic Programming?

Genetic Programming (GP) is a type of evolutionary computation, a subset of machine learning. Inspired by biological evolution, GP uses random mutation, crossover, a fitness function, and multiple generations of evolution of a population of computer programs to resolve a user-defined task. At the foundation level, GP discovers relationships between data features—correlations between measurements of the real world.

Long before the advent of gene sequencing or machine learning, taxonomists conducted precise measurements of physical plants and animals in order to differentiate species. In 1936, biologist R.A. Fisher worked to differentiate species of Iris flowers using data generated by Edgar Anderson. In his paper "The use of multiple measurements in taxonomic problems", Fisher states "When two or more populations have been measured in several characters, xl, ..., x8, special interest attaches to certain linear functions of the measurements by which the populations are best discriminated." Using Genetic Programming, we readily discover the mathematical expressions that cleanly separate the 3 Iris species using just 50 data points each. This Iris multivariate dataset is a simple yet classic, must-solve problem for all Machine Learning developers of classification algorithms, and is included with Karoo GP as the default Classification dataset.

In general, genetic programming can be used to discover a relationship between features in data (symbolic regression) or to group data into categories (classification). Both of these employ a training (learning) and test (validation) phase, after which the resulting function (mathematical expression) may be employed in a live data stream for realtime regression or classification analysis, accordingly.



The Karoo GP Interfaces

2-in-1: Desktop and Server

The *Desktop* interface provides a Text User Interface (TUI) for configuring, launching, monitoring, and testing your Karoo GP runs. There are 5 choices for how you monitor the evolutionary process:

- Generational (g): pauses after each generation is complete
- Interactive (i): pauses with the completion of each section (e.g. tournament, gene pool, genetic operators)
- DeBug (db): displays the internal workings of the genetic operators
- Minimal (m): displays only the multivariate expression of each tree
- Silent (s): displays only the summary of each generations

If you want to make additional configurations to the run before starting, select (g)enerational and then "?" to present the menu options. When you are ready to go, you can switch to (m)inimal mode and then press ENTER one more time. To launch the Desktop interface:

```
$ python karoo_gp.py
```

Without arguments, Karoo GP defaults to the datasets located in karoo_gp/files/ as determined by the fitness function (kernel) selected, eg: a Classification kernel loads data_CLASSIFY.csv.

If you include *only* a path to your own dataset:

```
$ python karoo_gp.py /[path]/[to_your]/[filename].csv
```

Karoo will yet present the Desktop interface, but will load your dataset. You must choose "Classification" or "Regression" according to the type of run you desire to conduct.

The *Server* interface supports immediate and chron scripted runs through which you can explore a series of tree depths, population sizes, or genetic operator configurations. If you append 2 or more command line arguments, Karoo will automatically enter Server mode. Keep in mind that you must include a pointer to data, as follows:

```
$ python karoo_gp_main.py -ker c -typ r -bas 4 -fil /[path]/[to_your]/[filename].csv
```

You can set your preferred default configuration parameters in the file karoo_gp_server.py such that without any command-line arguments, Karoo GP Server will repeat a run for comparison to priors. Command-line arguments then override these default settings, as follows:

argument	**options**	**description**
-ker	[r,c,m]	kernel: (r)egression, (c)lassification, or (m)atching
-typ	[f,g,r]	tree type: (f)ull, (g)row, or (r)amped half/half
-bas	[3...10]	base tree depth for the initial population
-max	[3...10]	maximum tree depth for the entire run
-min	[3...2^(base_tree_depth +1) - 1]	minimum number of nodes
-pop	[10...1000]	maximum population
-gen	[1...100]	number of generations
-tor	[7 for each 100 rcm]	qty of trees selected for the tournament
-fil	/[path]/[to_your]/[filename].csv	load an external dataset

In addition, you can change the balance of the genetic operators with -evr, -evp, -evb, and -evc.

```
Tree 14 yields (sym): a - a/c - b*c**2 + b*c + b - 2*c - 1/c + c**2/b + c/b + 1/b
Tree 15 yields (sym): -a**3 - a*b**2 + a/(b*c**2) - 4*b + 3*c + 1 - 1/c
Tree 16 yields (sym): a**3*c + a*b + a*b/c - a + b*c + 3*c + 1 - 1/c + c/b
Tree 17 yields (sym): -a**2*c/b - 2*a + a/b**2 - b**2 - 2*b*c - 3*b + c/b - c/a
Tree 18 yields (sym): a*b**2 + a*c - a - a/c - b**2 - b + 2*b/c - c - c/b - b**2/a + c**2/a
Tree 19 yields (sym): -2*a**2 + a*b - b + 2*c - 2 + 1/b - c/b**2 + c/a + 1/a
Tree 20 yields (sym): -a**2*c - a**2/c - a - a/b - b**3*c + b**2 + 2*b + c**2 + c - 1 - c**3/a
Tree 21 yields (sym): -a**2*c - a*b + a - a*c/b - a*c/b**2 - c**2 + c
Tree 22 yields (sym): a**2 - a*b*c - a*b + a*c + b*c**3 - 2*b/c - c**2 + 4*c - 1 + b*c/a
Tree 23 yields (sym): a*b*c + a*b + a*b/c - a + b**2*c + b*c**3 + b - c + 1
Tree 24 yields (sym): -a*b*c + a*c**2 + a*c - 2*a - b*c + b + 4*c - 1 + b/(a*c) + 1/(a*c)
Tree 25 yields (sym): -a*b*c**2 + a*b*c - a*c + 2*a + 3*b - 3*c - 1 + 1/b - b**2/(a*c)
Tree 26 yields (sym): -a*b*c + 2*a*b - b**2*c - b - c**2 - 3*c + 2 - c**2/b - b*c/a
Tree 27 yields (sym): -a**2 - a/c + a*c**2/b + a/b - a*c/b**3 + b*c - b + 2*c
```

First time through

The Karoo GP Desktop interface will prompt you for a series of inputs. Your first time through, enter Play mode in order to learn the functions of Karoo GP and to learn to visualise a GP tree, as follows:

1. Select Play mode.
2. Select a Full tree.
3. Select a depth of 1.

    Depth 1: 1 operator, 2 variables, eg: a / c

    Depth 2: 3 operators, 4 variables, eg: a + b – 2 * c

    Depth 3: 7 operators, 8 variables, eg: c + b * c^3 / a + c / a

```
Tree Depth: 0 of 1

NODE: 1
 type: root
 label: /      parent node:
 arity: 2      child node(s): 2 3

      Tree Depth: 1 of 1

      NODE: 2
       type: term
       label: a       parent node: 1
       arity: 0       child node(s):

      NODE: 3
       type: term
       label: c       parent node: 1
       arity: 0       child node(s):

     Tree 1 yields (raw): (a)/(c)
     Tree 1 yields (sym): a/c
```

Take a look the GP tree represented on-screen. There are 3 nodes composed of 2 terminals (variables) and 1 function (operator). Select Play mode a few more times, playing with larger trees and both Full and Grow methods. You will find that the branches of Full trees always reach the bottom, where the branches of Grow trees may terminate early, with a terminal always the final node. With some practice, you can read the tree as printed on screen. Future versions of Karoo GP will include an option to generate an image (similar to the cover page).

Second time through

Run Karoo GP again, but this time simply press ENTER at each of the queries. Sit back and watch as Karoo GP works to resolve a simple relationship between 3 variables which represent 3 columns of data: a + b + c = s.

We know the answer, but Karoo GP must discover a solution through an evolutionary process. As GP is based on random number generation, there is a chance that in the default 10 generations it will not work, or it might find a relationship between the columns of data other than that which was expected.

By configuring a run for deeper trees, or by increasing the minimum number of nodes required, you can encourage the evolutionary process to discover more complex solutions.

When done, you will be presented with *pause*. Type ? and ENTER to review the menu:

1. Type *l* and ENTER to view all trees which provide the correct solution.
2. Type *p*, ENTER, and then the unique number of any tree in the list, to print that tree to screen.
3. The solution will be listed in both its *raw* and Sympyfied format. Remember that multiple trees may carry the same or more than one solution to the same problem.
4. You may adjust the (b)alance of operators, (cont)inue the evolutionary process, (e)val the trees, or (q)uit.

```
    Tree 93 yields (sym): a + 2*b + 1
    Tree 94 yields (sym): a + 2*b + c/b
    Tree 95 yields (sym): 2*b + c/b**2
    Tree 96 yields (sym): a + a/c + b + c
    Tree 97 yields (sym): a + b + c + 1
    Tree 98 yields (sym): a + a/c + b + c
    Tree 99 yields (sym): a + b + b/c
    Tree 100 yields (sym): a + 3*b

28 trees [ 1  2  3  4  5  6  7  8  9 10 14 18 21 25 37 38 39 49 50 55 59 64 74 76 84 85 91 93] offer the highest fitness scores.
```

```
Tree 14 yields (sym): a - a/c - b*c**2 + b*c + b - 2*c - 1/c + c**2/b + c/b + 1/b
Tree 15 yields (sym): -a**3 - a*b**2 + a/(b*c**2) - 4*b + 3*c + 1 - 1/c
Tree 16 yields (sym): a**3*c + a*b + a*b/c - a + b*c + 3*c + 1 - 1/c + c/b
Tree 17 yields (sym): -a**2*c/b - 2*a + a/b**2 - b**2 - 2*b*c - 3*b + c/b - c/a
Tree 18 yields (sym): a*b**2 + a*c - a - a/c - b**2 - b + 2*b/c - c - c/b - b**2/a + c**2/a
Tree 19 yields (sym): -2*a**2 + a*b - b + 2*c - 2 + 1/b - c/b**2 + c/a + 1/a
Tree 20 yields (sym): -a**2*c - a**2/c - a - a/b - b**3*c + b**2 + 2*b + c**2 + c - 1 - c**3/a
Tree 21 yields (sym): -a**2*c - a*b + a - a*c/b - a*c/b**2 - c**2 + c
Tree 22 yields (sym): a**2 - a*b*c - a*b + a*c + b*c**3 - 2*b/c - c**2 + 4*c - 1 + b*c/a
Tree 23 yields (sym): a*b*c + a*b + a*b/c - a + b**2*c + b*c**3 + b - c + 1
Tree 24 yields (sym): -a*b*c + a*c**2 + a*c - 2*a - b*c + b + 4*c - 1 + b/(a*c) + 1/(a*c)
Tree 25 yields (sym): -a*b*c**2 + a*b*c - a*c + 2*a + 3*b - 3*c - 1 + 1/b - b**2/(a*c)
Tree 26 yields (sym): -a*b*c + 2*a*b - b**2*c - b - c**2 - 3*c + 2 - c**2/b - b*c/a
Tree 27 yields (sym): -a**2 - a/c + a*c**2/b + a/b - a*c/b**3 + b*c - b + 2*c
```

Subsequent runs

When you are ready to dive in a bit further, take a few minutes to learn about each of the following:

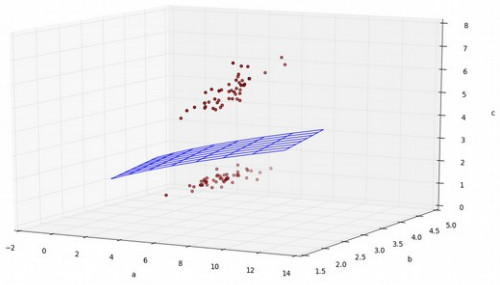
1 – Fitness Function (Kernel) Select

When prompted with a choice of (c)lassify, (r)egression, (m)atching, or (p)lay:

Regression is a minimisation function, meaning it seeks a fitness score with the *lowest* number. By default, this kernel will attempt to solve Kepler's 3rd Law of Planetary Motion. As the correct solution is t^2 / r^3, the challenge is for GP to not fall to a local minimum of *t/t or r/r. Yo*u can experiment with both deeper trees and increasing the minimum number of nodes. Or, let Karoo GP run for 20, 30, ... 50 or more generations and random mutation might just discover the correct solution.

To learn more, visit: www.physicsclassroom.com/class/circles/Lesson-4/Kepler-s-Three-Laws

Classification is a maximisation function, meaning it seeks a fitness score with the *highest* number. By default, this kernel will attempt to solve the Iris flower problem. This is a machine learning classic built upon data generated before the dawn of DNA classification, when botanists used microscopes and calipers to measure the features of plants and animals. While Kepler's Law has one solution, the Iris problem has many. If you plot one of the GP generated expressions over a scatter plot of the data (see tools/) you should find a clean separation of 2 or 3 species.



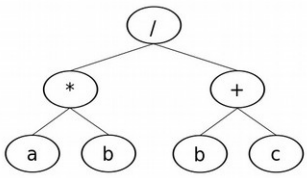
To learn more, visit: archive.ics.uci.edu/ml/datasets/Iris

Karoo GP supports multiclass classification from 2 to *n* classes, with automated scaling. See *Additional Scripts* (below) to learn more, and to play with the included karoo_multiclassifier.py found in the tools/ directory.

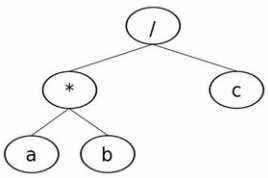
Match will attempt to discover a relationship between variables which produces an exact match, as with *a + b + c = s*, as noted in *Second Time Through* (above). This is the simplest of the fitness functions to experiment with on your own, but has limited realworld application. Try increasing tree depth and the minimum number of nodes. See *Using Your Own Data* at the bottom of this guide to learn how to replace the default data with your own.

2 – Type of Tree

All Full tree branches reach the maximum depth. These trees are less likely to solve problems as they are not as flexible in their solution space. For example, if the desired solution is a + b + c (5 elements) but the Full tree is confined to 15 elements, some will be forced to cancel each other in order to arrive to the desired solution. However, Full trees do contribute higher order expressions.



Full method GP tree



Grow method GP tree

Grow trees have unbalanced branches, meaning some branches reach the maximum depth while others do not. Grow trees are more likely to find simpler solutions, but may not discover improved solutions in a higher order space.

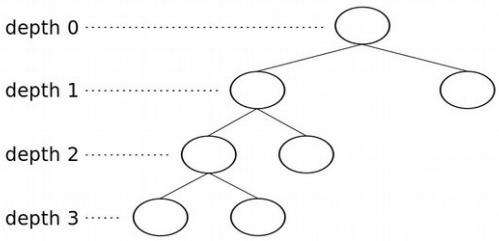
Originated by John Koza, *Ramped Half/Half* initialises the first population with a 50/50 split of Full/Grow methods, and a spread of depths from min to max. The Grow method is then applied for each subsequent population. This is the most popular method as it injects a higher degree of diversity into the initial population.

```
Tree 14 yields (sym): a - a/c - b*c**2 + b*c + b - 2*c - 1/c + c**2/b + c/b + 1/b
Tree 15 yields (sym): -a**3 - a*b**2 + a/(b*c**2) - 4*b + 3*c + 1 - 1/c
Tree 16 yields (sym): a**3*c + a*b + a*b/c - a + b*c + 3*c + 1 - 1/c + c/b
Tree 17 yields (sym): -a**2*c/b - 2*a + a/b**2 - b**2 - 2*b*c - 3*b + c/b - c/a
Tree 18 yields (sym): a*b**2 + a*c - a - a/c - b**2 - b + 2*b/c - c - c/b - b**2/a + c**2/a
Tree 19 yields (sym): -2*a**2 + a*b - b + 2*c - 2 + 1/b - c/b**2 + c/a + 1/a
Tree 20 yields (sym): -a**2*c - a**2/c - a - a/b - b**3*c + b**2 + 2*b + c**2 + c - 1 - c**3/a
Tree 21 yields (sym): -a**2*c - a*b + a - a*c/b - a*c/b**2 - c**2 + c
Tree 22 yields (sym): a**2 - a*b*c - a*b + a*c + b*c**3 - 2*b/c - c**2 + 4*c - 1 + b*c/a
Tree 23 yields (sym): a*b*c + a*b + a*b/c - a + b**2*c + b*c**3 + b - c + 1
Tree 24 yields (sym): -a*b*c + a*c**2 + a*c - 2*a - b*c + b + 4*c - 1 + b/(a*c) + 1/(a*c)
Tree 25 yields (sym): -a*b*c**2 + a*b*c - a*c + 2*a + 3*b - 3*c - 1 + 1/b - b**2/(a*c)
Tree 26 yields (sym): -a*b*c + 2*a*b - b**2*c - b - c**2 - 3*c + 2 - c**2/b - b*c/a
Tree 27 yields (sym): -a**2 - a/c + a*c**2/b + a/b - a*c/b**3 + b*c - b + 2*c
```

3 – Base Tree Depth

This is the tree depth for the *initial* population. By default, this is also the Maximum Tree Depth (below), unless otherwise user specified. If the base and maximum tree depths are equal, this will inhibit bloat, but also restrict potential, more complex solutions.

The depth of a tree is illustrated to the right, and relates to the maximum possible number of nodes, as follows:



nodes = 2^(depth + 1) – 1

4 – Maximum Tree Depth

Karoo is unique from traditional tree-based GP in that it incorporates a user defined maximum tree depth, thereby restricting program bloat. However, deeper trees present opportunity for more complex solutions, and they enable the inclusion of a greater number of features (columns in your .csv). As depth 3 enables up to 15 nodes, and depth 5 enables up to 63 nodes, quite a bit of growth is possible with trees just 2-3 depths greater. Or, you can set the maximum to 10 (2047 possible nodes) and learn if a larger solution is in fact more able to resolve your data.

5 – Minimum Number of Nodes

If the Maximum Tree Depth is the *ceiling*, then this is the *floor*. The minimum number of nodes (both operators and terminals) defines the simplest expression allowed. For example, the correct solution to Kepler's 3rd Law of Planetary Motion is t * t / r * r * r which has 9 elements (nodes). In Karoo GP, the gene pool from which the tournament selection operates is built only from those trees which meet the minimum number of nodes criterion. This is very useful in solving the default problem for the Regression kernel, as GP tends to the simpler t / t.

NOTE: *If you set the minimum number of nodes too high, you may invoke elitism in the population, killing off simpler solutions or even the entire population.*

6 – Number of Trees

100 trees per population is a good place to start. However, larger populations offer a greater diversity of possible solutions. Feel free to experiment with larger populations.

7 – Number of Generations

For simpler problems with less than a dozen features, 10 generations will often give you a sense of whether or not Karoo GP is on the right track. With more complex problems, 20, 30, … 50 or more generations may be required for the random process of evolution to generate an improved tree which provides the desired solution.

You may (cont)inue with subsequent generations when all generations have run their course (see bottom).

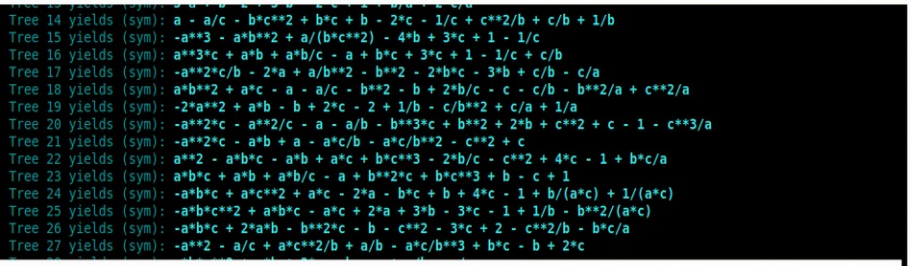
8 – Display Mode

The display modes (i)nteractive, (m)inimal, (g)eneration, and (s)ilent are the means by which you interact with and monitor the evolutionary process of Karoo GP. They do not affect the outcome of the process itself.

(i)nteractive – monitor the fitness function, tournament selection, and population of the gene pool from which the next generation will be selected. This mode will *pause* with each step of the evolutionary process.

(m)inimal - runs through the entire evolution, start to finish without *pause*. However, it does display each tree as it is evaluated, which is quite entertaining (if you are into that sort of thing).

(g)eneration – *pause* with the end of each generation, allowing you to make adjustments once per generation.



(s)ilent - designed to run remotely on a server with limited overhead. This mode presents the results of each generation just to let you know it is still alive 'n cranking through data while you are engaged in data reductions, Facebook, or a Star Wars role playing game.

(d)e(b)ug - for those of you who want to watch the evolutionary process at the intimate level, this is a fully transparent window to the inner workings of tree-by-tree Mutation and Crossover.

You can switch between modes at any *pause*. However, once you are in *minimal* or *silent* modes, you will no longer have opportunity to modify the evolutionary parameters until all generations are complete. However, you may continue the evolutionary process and Karoo GP will pick up where you left off. Simply enter ‘cont’ followed by the number of subsequent generations, and off you go. If new parameters are desired, remember to set all configurations before you continue.

What You See On-Screen

The *pause* menu

Be certain to play around with all the options available to you in *pause*. “?” or “help” followed by ENTER will display all available options. You can change the level of interaction, minimum number of nodes, number of CPU cores; evaluate and print, and more.

The most fit trees

At the end of each generation of population evolution, a list of bold numbers is presented on-screen. These are the trees GP has found to offer the best overall fitness scores in the latest generation. At each *pause*, you may (l)ist, (p)rint, and (e)valuate any given tree, and again at the end of any given run.

The leading Trees and their associated expressions are:

1 : 2*a - b + 2*c

2 : a + 2*b + 1

5 : a + 2*b + 1

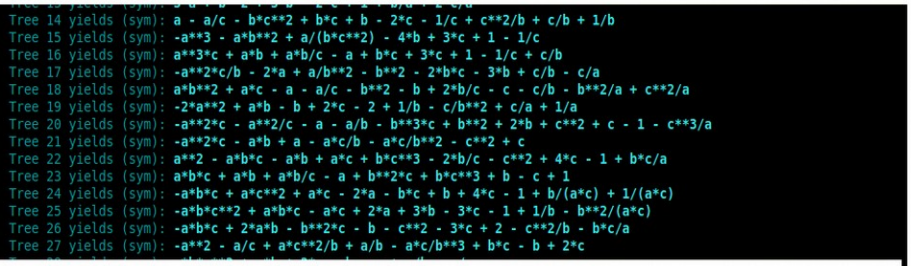
6 : a + b + c

10 : a + 2*b + 1

12 : a + 2*b + 1

21 : 2*a - b + 2*c

94 : a + 2*b + 1



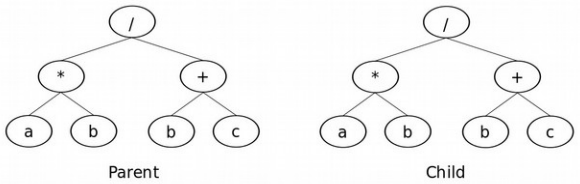
The Evolutionary Operators

There are 4 evolutionary operators applied in Karoo GP. The foundation for these evolutionary operators was derived from the "Field Guide to Genetic Programming" * by Riccardo Poli, William Langdon, and Nicholas McPhee, with contributions by John Koza. Using the (b)alance option in the *pause* menu, you can set the ratio of the operators, such that combined they equal 100%.

In the following, *tournament selection* refers to the random selection of a number of trees (default 10) whose fitness scores are compared. The tree with the highest score is moved to the next generation through an evolutionary operation. We use tournament selection, not a top-to-bottom evaluation of the entire population, else we risk *elitism*, premature convergence on what may not be the best overall solution.

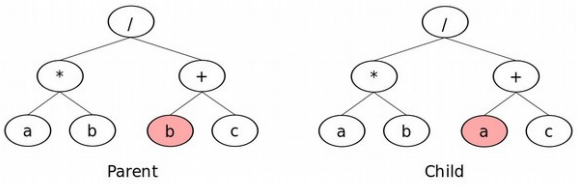
Reproduction

Through tournament selection, a single tree from the prior population is copied without mutation to the next generation. In the biological world, this is analogous to a member of a population entering the gene pool of the subsequent (younger) generation.



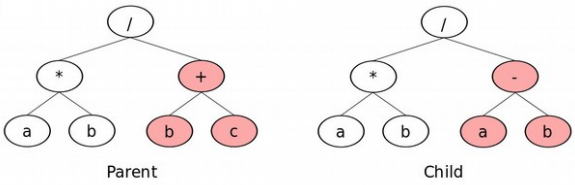
Point Mutation

Through tournament selection, a copy of a tree from the prior population mutates a single node before being added to the next generation. In the biological world, this may be analogous to asexual reproduction, that is, a copy of an individual with a minor mutation. In this method, a single point is selected for mutation while maintaining function nodes as operators and terminal nodes as terminals. The size and shape of the tree will remain identical.



Branch Mutation

Through tournament selection, a copy of a tree from the prior population mutates before being added to the next generation. In the biological world, this may be analogous to asexual reproduction, that is, a copy of an individual but with a potentially *substantial* mutation. Unlike Point Mutation, in this method an entire branch is selected. If the evolutionary run is designated as Full, the size and shape of the tree will remain identical, each node mutated sequentially, where functions remain functions and terminals remain terminals. If the evolutionary run is designated as Grow or Ramped Half/Half, the size and shape of the tree may grow smaller or larger, but it may not exceed the maximum depth defined by the user.

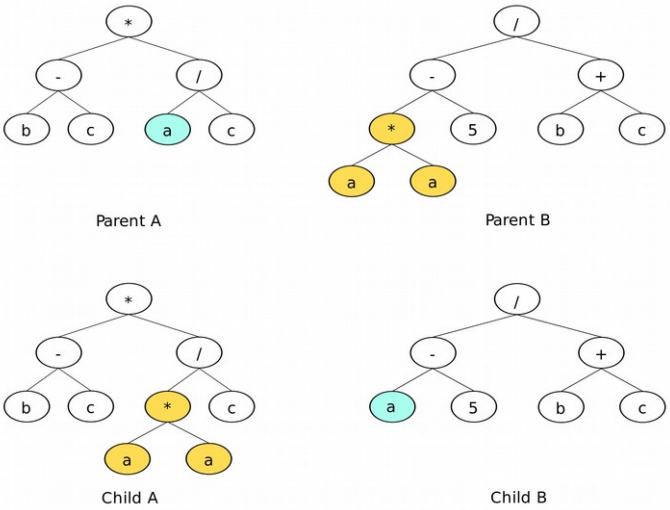


* The Field Guide and many other books about GP are listed at www.geneticprogramming.com

```
Tree 14 yields (sym): a - a/c - b*c**2 + b*c + b - 2*c - 1/c + c**2/b + c/b + 1/b
Tree 15 yields (sym): -a**3 - a*b**2 + a/(b*c**2) - 4*b + 3*c + 1 - 1/c
Tree 16 yields (sym): a**3*c + a*b + a*b/c - a + b*c + 3*c + 1 - 1/c + c/b
Tree 17 yields (sym): -a**2*c/b - 2*a + a/b**2 - b**2 - 2*b*c - 3*b + c/b - c/a
Tree 18 yields (sym): a*b**2 + a*c - a - a/c - b**2 - b + 2*b/c - c - c/b - b**2/a + c**2/a
Tree 19 yields (sym): -2*a**2 + a*b - b + 2*c - 2 + 1/b - c/b**2 + c/a + 1/a
Tree 20 yields (sym): -a**2*c - a**2/c - a - a/b - b**3*c + b**2 + 2*b + c**2 + c - 1 - c**3/a
Tree 21 yields (sym): -a**2*c - a*b + a - a*c/b - a*c/b**2 - c**2 + c
Tree 22 yields (sym): a**2 - a*b*c - a*b + a*c + b*c**3 - 2*b/c - c**2 + 4*c - 1 + b*c/a
Tree 23 yields (sym): a*b*c + a*b + a*b/c - a + b**2*c + b*c**3 + b - c + 1
Tree 24 yields (sym): -a*b*c + a*c**2 + a*c - 2*a - b*c + b + 4*c - 1 + b/(a*c) + 1/(a*c)
Tree 25 yields (sym): -a*b*c**2 + a*b*c - a*c + 2*a + 3*b - 3*c - 1 + 1/b - b**2/(a*c)
Tree 26 yields (sym): -a*b*c + 2*a*b - b**2*c - b - c**2 - 3*c + 2 - c**2/b - b*c/a
Tree 27 yields (sym): -a**2 - a/c + a*c**2/b + a/b - a*c/b**3 + b*c - b + 2*c
```

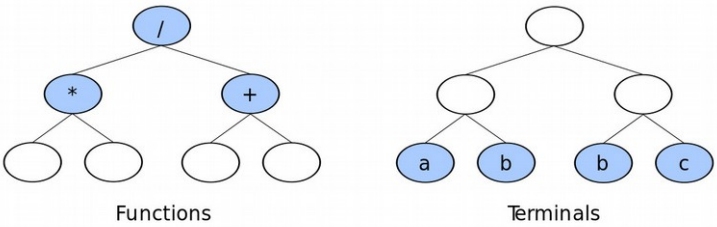
Crossover

Through tournament selection, 2 trees are selected as *parents* to produce 2 *offspring*. Within each parent tree a branch is selected. For child A, parent A is copied, with its selected branch deleted. Parent B's branch is then copied to the former location of parent A's branch, and inserted (grafted). This is reversed for child B. The size and shape of the offspring may be smaller or larger than either of the parents, but may not exceed the maximum depth defined by the user.



This process combines genetic code from trees which were chosen by the tournament process as having a higher fitness than the average population. Therefore, there is a higher probability their offspring will provide an improved fitness.

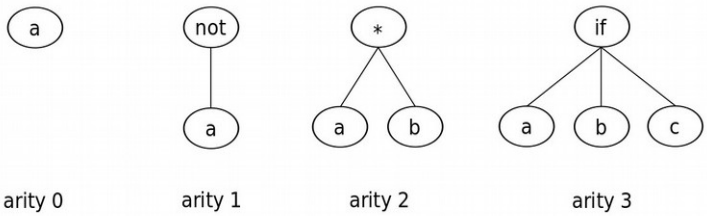
According to the literature, crossover is the most commonly applied evolutionary operator at 70-90%.

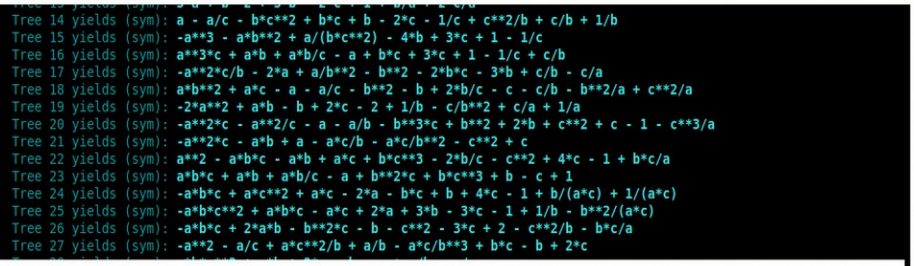


Functions (Operators)

The operators provide the functionality of the multivariate expression, the relationships between the operands and the features they represent. The most commonly used operators are arithmetic [+, -, *, /, **], trigonometric [cos, sin, tan], Boolean [and, or, not], and abs, log, sign, and sqrt. A complete list of all supported operators is available at karoo_gp/files/templates/operators_list.txt

All operators presented to Karoo GP must be followed by an *arity*, the number of variables or constants each operator expects to work with. For example, * has an arity of 2, meaning it expects something before and after itself, as in the expression *a * b*. Some examples of arity are given here:





Karoo can readily discover powers of 2, 3, or 4 by multiplying a given operand by itself that many times. However, a power of 7, 10, or more would require substantially larger trees and likely many more generations. By providing the operator ** you are reducing the computational overhead as Karoo will be able to test higher order expressions with less exploration of the solutions space.

Square root too can be discovered through * and /, but as with ** there is a computational overhead to GP's discovery of this solution. Therefore *sqrt* may be introduced if anticipated by your understanding of the underlying principals of the dataset.

As presented in karoo_gp/files/templates/operators_list.txt, [cos, sin, exp, sqrt, log] must be preceded by another operator, as shown below. So, you will likely want to include all 4 of the arithmetic operators to provide a balanced set, as follows:

```
+ cos,2
- cos,2
* cos,2
/ cos,2
```

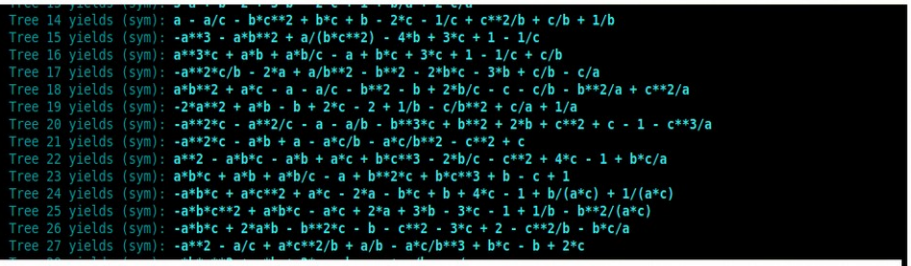
NOTE: *There is no space between the operator, the comma, and the arity.*

The operators applied to a given Karoo GP run are designated by you. The name of the .csv file which contains your desired operators must match the name of the fitness function (kernel) chosen. For example, the Classification kernel will auto-load /files/operators_CLASSIFY.csv.

It is important to note that the selection of any operator from this list is entirely random. The quantity of times any given operator appears relates to its chance of being selected, and therefore biases the outcome of the evolutionary process. If you have a single trigonometric operator which is represented 4 times, then you may desire to provide each of the arithmetic operators 4 times as well.

An example of the contents of a potential operators_[kernel].csv file follows:

```
operator, arity
+,2
-,2
*,2
/,2
+,2
-,2
*,2
/,2
+,2
-,2
*,2
/,2
+ cos,2
- cos,2
* cos,2
/ cos,2
```



Terminals (Operands)

Karoo GP automatically extracts the terminals (variables and constants) from the top of the data .csv file which you have provided. This enables you to name each column of your data to represent that particular feature. Later, when you read a GP derived expression, you will see how it relates to the features and original data.

Variables

If you are attempting to predict the weather using historical data[1] you might have something like this:

jd	sat	saz	hum	tmp	wm	wd	s
0	0.9292	1	0.2613	0.549	0.3956	0.0837	0
0.0092	0.9027	0.9865	0.2368	0.5862	0.4377	0.1328	0
0.0184	0.8761	0.9697	0.195	0.5908	0.4048	0.0913	0
0.0276	0.8496	0.9562	0.2021	0.6077	0.1905	0.1159	0
0.0368	0.8142	0.9461	0.2076	0.6156	0.2106	0.1221	0
0.0461	0.7788	0.9327	0.1821	0.611	0.185	0.0999	0
0.0553	0.7522	0.9226	0.1993	0.6099	0.152	0.3873	0
0.0645	0.7168	0.9125	0.2109	0.6043	0.2271	0.1416	0

Where “jd” is the variable for “Julian Day” (time), “sat” for the Sun's altitude, “saz” for the Sun's azimuth, “hum” for relative humidity, “tmp” for temperature, “wm” for the magnitude of the wind, and “wd” for the direction of the wind. The final column, “s” is the solution, that is, what you are are solving for. In this case, we are applying a regression algorithm, with a minimisation function approaching zero in order to discover a correlation between the given variables.

Constants

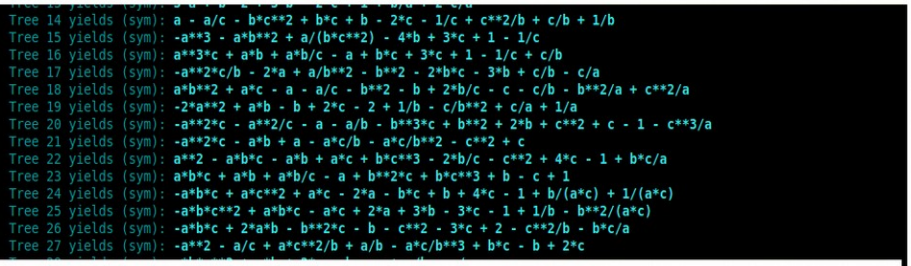
You can inject constants into the mix. In the current implementation of Karoo GP, constants are added at the top of unique columns in the data .csv. The entire column associated with each constant is then filled with zeros. These zeros are space fillers which have no impact on the processing of the data or operator.

For example, you might add a set of constant values [-5,-4,-3,-2,-1,1,2,3,4,5] which are balanced across the origin. Or, if you have a sense of how the variables need to be weighted, you may choose a particular set of constants. The same weather dataset above could include 5 constants expressed as decimal fractions, as the predicted variations in weights applied to this normalised data is relatively small.

jd	sat	saz	hum	tmp	wm	wd	0.1	0.2	0.3	0.4	0.5	s
0	0.9292	1	0.2613	0.549	0.3956	0.0837	0	0	0	0	0	0
0.0092	0.9027	0.9865	0.2368	0.5862	0.4377	0.1328	0	0	0	0	0	0
0.0184	0.8761	0.9697	0.195	0.5908	0.4048	0.0913	0	0	0	0	0	0
0.0276	0.8496	0.9562	0.2021	0.6077	0.1905	0.1159	0	0	0	0	0	0
0.0368	0.8142	0.9461	0.2076	0.6156	0.2106	0.1221	0	0	0	0	0	0
0.0461	0.7788	0.9327	0.1821	0.611	0.185	0.0999	0	0	0	0	0	0
0.0553	0.7522	0.9226	0.1993	0.6099	0.152	0.3873	0	0	0	0	0	0
0.0645	0.7168	0.9125	0.2109	0.6043	0.2271	0.1416	0	0	0	0	0	0

Future versions of Karoo GP will include constants as defined by the user via the interface.

[1] data from the SALT weather monitoring station and weather prediction software developed by Dr. Stephen Potter, head astronomer, South African Astronomical Observatory; Karoo GP is used to determine the weights



Data Management

Default Data Format

While Karoo GP can be recoded to work with any data format, by default, it anticipates .csv (comma separated values) files as readily exported from spreadsheet applications. Each row in a data .csv file is a data point (eg: a stamp in a time series), with 2 or more columns which hold raw data or a values derived through *feature engineering* (more below).

In supervised machine learning, it is standard for the right-most column to be the solution, either a value for which a regression algorithm is working to approach, or a label for each of 2 or more classes. Karoo GP requires the header for this column to be labeled lower-case “s”.

With the Iris dataset (included with Karoo GP as a demonstration of multi-class classification), each row represents a single flower in the real world. The columns hold the measurements of parts of each flower. The right-most column holds the solution, in this case the classification [0,1,2] for each of the 3 species. No matter if you are working in biology, sociology, or astrophysics, the function of the rows and columns remain the same.

Loading your Dataset

Karoo GP is designed for easily repeatable experimentation. When you launch Karoo GP, data and function files associated with the user selected kernel are auto-loaded. For example, if you select “c” for classification, then “files/data_CLASSIFY.csv” (see *Functions*) “files/operators_CLASSIFY.csv” is loaded accordingly.

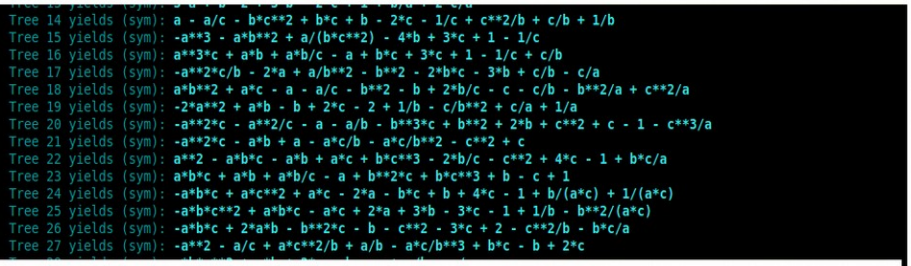
When you are ready to apply your own dataset, you can point Karoo GP to a properly prepared .csv which contains your data, as described in the Karoo GP Interface section for Desktop and Server, above.

NOTE: *No matter which dataset you load the operators associated with the selected kernel will be loaded. As such, you must first edit files/operators_[kernel].csv before your Karoo GP run.*

A few things to keep in mind:

1. Learn about the Sort and Normalise scripts (see *Tools* below) included with this package which will help you properly prepare your own dataset.
2. If you generate your own dataset using a spreadsheet application, be certain to save the original spreadsheet as .ods or .xls in addition to the .csv required by Karoo GP in order to preserve any formula, notes, or formatting for reference or later modification.
3. If you replace the contents of an existing .csv in the files directory with your own, pay attention to the following:
    - If editing the .csv using a text editor, do NOT add spaces between the variables and the comma.
    - Label each column with a variable. For example: *a,b,c* or *a1,a2,a3* or give each column a variable which represents the real-world measurement, such as *t* for time or *p* for pressure.
    - The right-most column must be labeled “s” (lower case, no spaces before or after).
4. Be *very* wary of your editor applying hidden characters in the .csv as these will cause Karoo to crash.
5. If sending a .csv file through email, wrap it in an archive (.zip, .tar.gz) else data corruption might occur.

NOTE: *Your .csv must conform to the format described in the above or bad things will happen.*



Data Size

In theory, the size of the .csv file is limited only by the amount of RAM in your computer. A practical run of GP is often against 10,000 rows of data and hundreds of features. With the addition of TensorFlow to Karoo GPU in early 2017, runs of many hours are reduced to just a few minutes. However, small datasets (dozens to thousands of total data points) will often run faster on a single core than on multiple cores or a GPU card. When the number of data points are measured in millions, execution of Karoo GP on multiple CPU cores and GPUs significantly.

Train vs Test

Using a SciKit Learn module, Karoo GP auto-splits all datasets greater than 10 lines into TRAINING and TEST in order to offer data on which to test that is unique from the training. However, if you present Karoo with 10 or fewer rows in your .csv, it automatically copies the same data into both TRAIN and TEST as it is assumed you are using Karoo GP in an instructional or experimental mode. This will not provide a valid run.

Tree Population Management

All trees generated for all populations of any given Karoo GP run are stored in runs/[datetime]/ as follows:

population_a.csv - includes the foundation population and all subsequent evolved populations. If you run 10 generations with 100 trees each, you will have a .csv file with 1000 trees.

population_b.csv - written only if you export a snapshot of the evolutionary process (at the *pause* menu) using the (w)rite command.

population_final.csv - the final generation of GP trees should, if all goes well, offer the most fit trees of all the generations. While this final generation is also included in population_a, this file is isolated in order to make it easier for you to locate and work with these solutions.

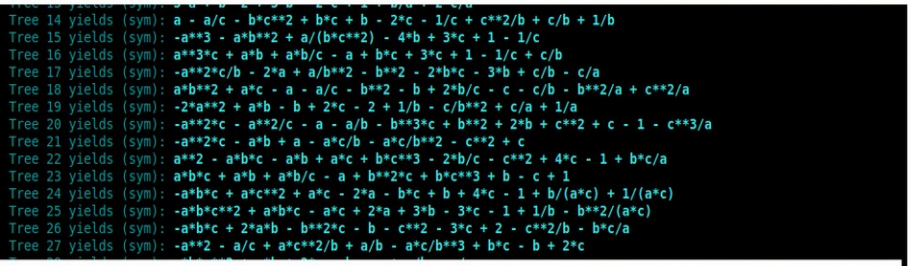
TREE_ID	100								
tree_type	g								
tree_depth_base	4								
NODE_ID	1	2	3	4	5	6	7	8	9
node_depth	0	1	1	2	2	3	3	3	3
node_type	root	func	term	func	func	term	term	term	term
node_label	*	*	b	+	-	b	b	c	a
node_parent		1	1	2	2	4	4	5	5
node_arity	2	2	0	2	2	0	0	0	0
node_c1	2	4		6	8				
node_c2	3	5		7	9				
node_c3									
fitness	5			19					

End-of-Run Logging

Karoo GP automatically logs the user-defined configuration parameters and an evaluation of the highest numbered (which is typically the highest scoring) tree in runs/[datetime]/ as log_config.txt and log_eval.txt accordingly. This enables the user to invoke scripts and chron jobs which launch multiple consecutive or parallel runs of Karoo without human intervention nor archiving of results.

Archived and Seed Populations

When you desire to restore a population, copy population_f to population_s.csv. From the *pause* menu, select “load” and immediately, population_s will *overwrite* population_a which is the foundation population for the next generation.



With Karoo GP you can load a former population or a hand-crafted, *seed populatio*n at any *pause*. This enables you to work with archived populations to carry forward with an evolutionary run with a new set of parameters to test a new hypothesis, work against a new dataset, or conduct a new series of validations. What's more, you can start with the same population and test any number of evolutionary parameter configurations over varying generations in order to determine which enables the most rapid convergence on a desired solution.

If you desire to load an archived (former but unchanged) population select the (g)eneration or (i)nteractive display mode. Just after launch, at the first *pause*, conduct the load, configure your run, and then press ENTER to continue.

Feature Engineering

If a dataset is a collection of raw data points, one or more values or measurements related to observation of the real world, then features are *transformed data*, values extracted from the observational data. According to Dr. Arun Aniyan, post-doctorate fellow at the Square Kilometer Array, South Africa, features are:

- Informative and non-redundant.
- Support subsequent learning and generalization.
- Reduce dimensionality (in most cases).
- Lead to better human interpretation of the overall data.

Feature engineering is the meat of machine learning, the most arduous and challenging aspect in which you become intimately familiar with your data in order to feed the machine learning algorithm *only* those features which give it the greatest chance of successful learning.

There are 3 basic processes within feature engineering: Extraction, Selection, and Construction, as follows:

Feature Extraction

Feature extraction is an automated method of working with observational data to build new, derived, informative and non-redundant features which lead to better interpretation of the overall data. Because many observations produce far too much data to be used directly in machine learning, feature extraction *reduces* dimensionality of the data while at the same time improves the success of the predictive models.

Common examples include audio, image, text, and tabular data (eg: the Iris dataset or time series data) with millions of data points. For tabular data, Principal Component Analysis (PCA) and unsupervised clustering methods might be employed. For image data, filters familiar to graphic designers such as edge detection or quantitative analysis of a particular colour might be employed.

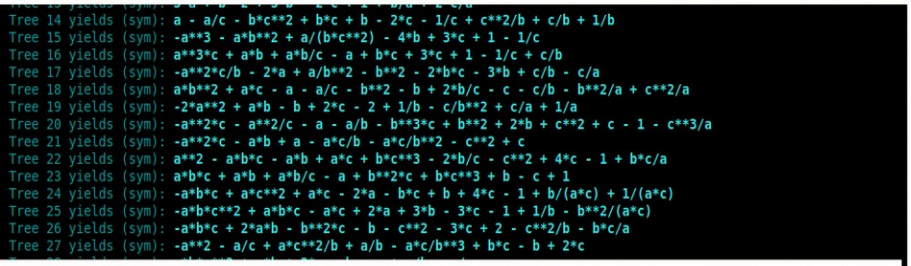
Key to feature extraction is that the methods are automatic (even if designed and constructed from simpler methods) which work to solve the problem of unmanageably high dimensional data.

SciKit Learn offers freely available methods for conducting Feature Extraction.

Feature Selection

Feature selection is the process of selecting a subset of relevant features which best describe the dataset as a whole, as some features may not be rich in discriminatory power. Feature selection helps reduce the dimensionality of the data in order that the machine learning algorithm is more readily able to learn. Common methods include comparing features via histograms or using genetic programming as a preprocessor to select which features to retain, and which to throw out.

SciKit Learn offers freely available methods for conducting Feature Selection in Python. Feature selection may also be conducted with Genetic Programming, as discussed by Bing Xue, et al, in this paper.



Feature Construction

Feature construction is similar to extraction in that you are working to build new, derived, informative and non-redundant features, but instead of an automated process, it is done manually. There are no standard means to this process, for it may differ for each dataset.

One method is to use Genetic Programming to build higher-order features, polynomial expressions which are in and of themselves demonstrations of relationships between two or more lower-level feature or raw data. This process is described by Soha Ahmed, et al, in "Multiple feature construction for effective biomarker identification and classification using genetic programming"

Develop your own Fitness Functions

Fitness functions lie at the heart of all machine learning. They may be very simple, classifying according to a value less than or equal to zero as class 0, or greater than zero as class 1. Or they may be very complex, applying external scripts which track trends in the classification process itself, even applying statistical analysis to determine how to adjust the fitness functions parameters over time.

Advanced users may modify existing fitness functions (kernels) or create their own. This guide does not provide a detailed guide, rather a pointer to the places in Karoo GP which require modification, as follows:

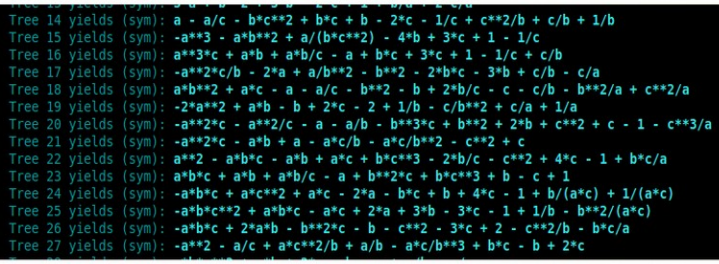
In karoo_gp_base_class.py:

1. In the method fx_data_load() modify the dictionaries: data_dict, func_dict, and fitt_dict
2. In fx_data_params_write() search for "[other]" and modify.
3. In fx_fitness_eval() search for "[other]" and modify.
4. Now build your new fitness function as both a Python *train* and *test* method. Search for "fx_fitness_test_[other]" to find the place-holder. While you can develop a new fitness function with straight Python coding, only with the application of the TensorFlow library will you be able to access the high performance of GPU processing. Refer to the other fitness functions for examples and to www.tensorflow.org for documentation and tutorials.
5. When complete, be certain to add a letter designation as a kernel options to the karoo_gp.py and to fx_karoo_pause() in karoo_gp_base_class.py where a template [other] is reserved.

Karoo Tools

A suite of simple, powerful data management tools are available from github.com/kstaats/karoo_tools

- Karoo Data Clean
- Karoo Data Norm
- Karoo Data Sort
- Karoo Multiclasssifier
- Karoo NPY Stack
- Karoo Pipeline



Experiment! Explore! *Evolve!*

You cannot damage Karoo GP by experimenting. If you screw up a data file, either Karoo GP or Python will complain (or just come to a screeching halt). Keep backups of your data and all original files. Save templates of those that work well. But most of all, *have fun!* And if you find any errors in the code (there may be a few), do not hesitate to contact me. I will do my best to respond quickly –kai

Notes