
Pygenetic Documentation

Release 1.0

Shreyas

May 09, 2019

Contents

1	Introduction	1
2	Features	3
2.1	Introduction	3
2.2	pygenetic package	5
2.3	README	5
2.4	Usage of pygenetic: An overview	7
2.5	1. Usage of GAEngine: the Low Level pygenetic GA API	7
2.6	2. Usage of <i>SimpleGA</i> : the High Level pygenetic GA API	11
3	Indices and tables	13

CHAPTER 1

Introduction

Pygenetic provides users a highly efficient and usable way to explore the problem solving ability of Genetic Algorithms. It seeks to reduce the task of solving a problem using genetic algorithms to just choosing the appropriate operators and values which are provided internally. Further, support is also provided for a user to input his own operators for variation or for solving more specific problems. Students, teachers, researchers, company employees / entrepreneurs can all use our genetic algorithm framework while experimenting with different Machine Learning Algorithms and observing performance.

- Presence of both High-Level(*SimpleGA*) and Low-Level API(*GAEngine*) which users can use as per need.
- Very generic API - Users can customize different part of the GA be it Evolution, Statistics, Different handlers, Chromosome Representations.
- Supports efficient evolution execution using Apache Spark. This is highly scalable as more workers can be deployed. Parallelization of fitness evaluation, selection, crossovers and mutations are taken care of.
- Supports Adaptive Mutation Rates based on how diverse the population is.
- Supports Hall of Fame(best ever chromosome) Injection so that the best chromosome isn't lost in later generations due to the selection method used.
- Supports Efficient Iteration Halt
- Supports Visualization of Statistics like max, min, avg, diversity of fitnesses, mutation rates. Users can also define custom statistics
- Supports usage of multiple crossovers and mutations in one GA execution to enhance diversity
- Supports Population Control which users can make use of in various research purposes
- Provides a bunch of Standard Selection, Crossovers, Mutations and Fitness Functions
- Provides continue evolve feature so users can continue from previous evolutions instead of starting all over again.
- Provides ANN Best Topology finder using GA functionality

Contents:

2.1 Introduction

Pygenetic provides users a highly efficient and usable way to explore the problem solving ability of Genetic Algorithms. It seeks to reduce the task of solving a problem using genetic algorithms to just choosing the appropriate operators and values which are provided internally. Further, support is also provided for

a user to input his own operators for variation or for solving more specific problems. Students, teachers, researchers, company employees / entrepreneurs can all use our genetic algorithm framework while experimenting with different Machine Learning Algorithms and observing performance.

2.1.1 Features

- Presence of both High-Level(SimpleGA) and Low-Level API(GAEngine) which users can use as per need.
- Very generic API - Users can customize different part of the GA be it Evolution, Statistics, Different handlers, Chromosome Representations.
- Supports efficient evolution execution using Apache Spark. This is highly scalable as more workers can be deployed. Parallelization of fitness evaluation, selection, crossovers and mutations are taken care of.
- Supports Adaptive Mutation Rates based on how diverse the population is.
- Supports Hall of Fame(best ever chromosome) Injection so that the best chromosome isn't lost in later generations due to the selection method used.
- Supports Efficient Iteration Halt
- Supports Visualization of Statistics like max, min, avg, diversity of fitnesses, mutation rates. Users can also define custom statistics
- Supports usage of multiple crossovers and mutations in one GA execution to enhance diversity
- Supports Population Control which users can make use of in various research purposes
- Provides a bunch of Standard Selection, Crossovers, Mutations and Fitness Functions
- Provides continue evolve feature so users can continue from previous evolutions instead of starting all over again.
- Provides ANN Best Topology finder using GA functionality

2.2 pygenetic package

2.2.1 Submodules

2.2.2 pygenetic.ANNEvolve module

2.2.3 pygenetic.ChromosomeFactory module

2.2.4 pygenetic.Evolution module

2.2.5 pygenetic.GAEngine module

2.2.6 pygenetic.Population module

2.2.7 pygenetic.SimpleGA module

2.2.8 pygenetic.Statistics module

2.2.9 pygenetic.Utills module


2.2.10 Module contents

2.3 README

<https://travis-ci.com/danny311296/pygenetic.svg?token=A3bcYHcDEvK23esetBsC&branch=master> (<https://travis-ci.com/danny311296/pygenetic>)

pygenetic is a Python Genetic Algorithm API which is User-Friendly as well as Generic in nature unlike most GA APIs which make a trade off between the two.

2.3.1 Motivation

 While some APIs like DEAP and many more recent ones which are very efficient and generic are less user friendly in nature, other APIs like genetics and other smaller ones which are the best in terms of user friendliness, they are less generic. This API intends to strike a balance - good in terms of both user friendliness and genericity.

2.3.2 Features

- Presence of both High-Level(*SimpleGA*) and Low-Level API(*GAEngine*) which users can use as per need.
- Very generic API - Users can customize different part of the GA be it Evolution, Statistics, Different handlers, Chromosome Representations.
- Supports efficient evolution execution using Apache Spark. This is highly scalable as more workers can be deployed. Parallelization of fitness evaluation, selection, crossovers and mutations are taken care of.
- Supports Adaptive Mutation Rates based on how diverse the population is.
- Supports Hall of Fame(best ever chromosome) Injection so that the best chromosome isn't lost in later generations due to the selection method used.

- Supports Efficient Iteration Halt
- Supports Visualization of Statistics like max, min, avg, diversity of fitnesses, mutation rates. Users can also define custom statistics
- Supports usage of multiple crossovers and mutations in one GA execution to enhance diversity
- Supports Population Control which users can make use of in various research purposes
- Provides a bunch of Standard Selection, Crossovers, Mutations and Fitness Functions
- Provides continue evolve feature so users can continue from previous evolutions instead of starting all over again.
- Provides ANN Best Topology finder using GA functionality

2.3.3 Installation

pygenetic is published on pypi(<https://pypi.org/project/pygenetic/>) and can be easily installed by:

```
$ pip3 install pygenetic
```

2.3.4 Tests

The various tests are present in the *tests/* directory. The main API tests can be tested by:

```
$ pytest tests/modules
```

2.3.5 Usage

Refer *examples* and ReadTheDocs(<https://pygenetic.readthedocs.io/en/latest>) More tutorials coming soon. . .

2.3.6 GA Online Execution

Install python *flask* and run

```
$ python3 flask/views.py
```

Input all the various fields needed for the GA. You can run the GA online and get the best 5 chromosomes of each generation followed by statistics. You can also download the equivalent pygenetic code based on all user inputs in the form

2.3.7 Authors

- Bharatraj S Telkar (<https://github.com/BharatRajT>)
- Daniel Isaac (<https://github.com/danny311296>)
- Shreyas V Patil (<https://github.com/pshreyasv100>)

2.3.8 Special Mentions

- Special thanks to Ganesh K, Rahul Bhardwaj and Hardik Surana who lended their UI made for their Design Patterns project (<https://github.com/ganesh-k13/GOF-Templates>) as an intial template for us to work on for our Web GUI.
- Special thanks to our Project Guide Prof.Chitra G M

2.3.9 License: MIT

2.4 Usage of pygenetic: An overview

2.5 1. Usage of GAEngine: the Low Level pygenetic GA API

2.5.1 1.1 Creating a Chromosome Factory

Chromosome Factories specify how the chromosome for the GA is to be created.

pygenetic supports two types of ChromosomeFactories * ChromosomeRegexFactory: for creating chromosomes whose genes follow a particular regex * ChromosomeRangeFactory: for creating chromosomes whose genes are between some numeric interval

1.1.1 Usage of ChromosomeRangeFactory

```
>>> from pygenetic import ChromosomeFactory
>>> factory = ChromosomeFactory.ChromosomeRangeFactory(minValue=1,
                                                       maxValue=100,noOfGenes=8,duplicates=False)
```

This creates a factory to create chromosomes with 8 genes and those genes can take values between 1 and 100 with no duplicates

We can test if it creates chromosomes as expected by calling the *createChromosome* method of the factory

```
>>> factory.createChromosome()
[62, 24, 10, 84, 93, 40, 86, 87]
```

1.1.2 Usage of ChromosomeRegexFactory

```
>>> factory = ChromosomeFactory.ChromosomeRegexFactory(pattern='0|1|7',noOfGenes=10,
↳data_type=int)
>>> factory.createChromosome()
[7, 7, 7, 0, 0, 0, 7, 1, 1, 7]
```

This creates a factory to create chromosomes with 10 genes and those genes can take values from the regex *0|1|7* with the genes converted to an integer data type.

1.1.3 Custom Chromosome Factories

Users can easily define custom factories by subclassing ChromosomeFactory

```

>>> class CustomFactory(ChromosomeFactory.ChromosomeFactory):
...     def __init__(self, noOfGenes, input_list):
...         self.noOfGenes = noOfGenes
...         self.input_list = input_list
...     def createChromosome(self):
...         return random.sample(self.input_list, self.noOfGenes)
...
>>> factory = CustomFactory(noOfGenes=5, input_list=['duck', 'cow',
...         'monkey', 'giraffe', 'dog', 'cat', 'peacock', 'mice', 'sun'])
>>> factory.createChromosome()
['mice', 'giraffe', 'cow', 'dog', 'cat']

```

This factory creates a chromosomes whose values are taken from values given in an input list.

2.5.2 1.2 Defining the GA using GAEngine

We can now create the GAEngine which is responsible for running the GA.

It can easily created using the factory created earlier

```

>>> from pygenetic import GAEngine
>>> ga = GAEngine.GAEngine(factory=factory, population_size=100, fitness_type=('equal',
↪8),
...                         cross_prob=0.7, mut_prob = 0.1)

```

where *factory* is the **ChromosomeFactory** to be used in the GA *population_size* is the size of population to be used in the GA *fitness_type* is the fitness type which can be either *max*, *min* or (*equal*, <value-to-acheive>) *cross_prob* is the crossover probability *mut_prob* is the mutation probability

Other parameters of *GAEngine* include * *adaptive_mutation*: which is a Boolean which decides if adaptive mutation is to be used (default: True) * *population_control*: which is a Boolean which decides whether or not the GAEngine should ensure that the population size remains the same in every evolution iteration. This ensures that any error/issue in user's custom selection or evolution code doesn't cause population size to change. (default: False) * *hall_of_fame_injection*: which is a boolean used to carry out the injection of the best chromosome encountered so far in every 20 generations. (default: True) * *efficient_iteration_halt*: which is a boolean used to carry out *efficient_iteration_halt* optimization. It stops evolving if same best fitness value is encountered for 20 consecutive generations (default: True) * *use_pyspark*: which is a boolean used to decide if sequential execution is to be carried out or parallel execution on Apache Spark is to be carried out (default: False)

2.5.3 1.3 Crossovers, Mutations and Selection Functions

1.3.1 Basics

We should then add appropriate Crossovers, Mutations and Selection Functions for our GA execution

Many standard Crossovers, Mutations and Selection Functions are already available in *Utils* of *pygenetic* module. *Utils* contains the following 1. Selection - *random*, *best*, *tournament*, *roulette*, *rank* and *SUS* 2. Crossover - *distinct*, *onePoint*, *twoPoint*, *PMX* and *OX* 3. Mutation - *swap* and *bitFlip*

pygenetic supports more than one crossovers and mutations in one GA execution.

```

>>> from pygenetic import Utils
>>> ga.addCrossoverHandler(Utils.CrossoverHandlers.distinct, 4)
>>> ga.addCrossoverHandler(Utils.CrossoverHandlers.OX, 3)
>>> ga.addMutationHandler(Utils.MutationHandlers.swap, 2)

```

where the first parameter is the handler and the second parameter is the weightage to be given to that handler.

Note: for handlers which require parameters (eg: *tournament*), the parameters are passed after the weight parameter when added to the *GAEngine* object

```
from pygenetic import Utils # 4 is the weightage while 2 is the parameter of the handler(tournament size) >>>
ga.addCrossoverHandler(Utils.CrossoverHandlers.tournament, 4, 2)
```

We can also add selection handler in the same fashion

```
>>> ga.setSelectionHandler(Utils.SelectionHandlers.best)
```

1.3.2 Custom Handlers

Users can also define custom Crossovers, Mutations and Selection handlers where the function follows the prototype `>>> def custom_function(fitness_mappings, ga)` where *fitness_mappings* is a list of tuples where each tuple is of the form *(chromosome, fitness_score)*

ga is the entire *GAEngine* object which user would created. A user can access any detail regarding the GA over here. (Note: use *ga.population.members* to access current population members)

Note: Crossover and Mutation handlers should return a tuple of the two new children while Selection handlers should return a list of selected chromosomes.

It can be added as always

```
>>> ga.addCrossoverHandler(custom_function, 4)
>>> ga.addMutationHandler(custom_function2, 2)
>>> ga.setSelectionHandler(custom_function3)
```

Users can also define parameterized custom Crossovers, Mutations and Selection handlers where the function follows the prototype `def custom_function(fitness_mappings, ga, ...)` where *fitness_mappings* is a list of tuples where each tuple is of the form *(chromosome, fitness_score)*

ga is the entire *GAEngine* object which user would created. A user can access any detail regarding the GA over here. ... are any other arguments needed

It can be added as always

```
# where 4 is the weightage >>> ga.addCrossoverHandler(custom_function, 4, ...) >>>
ga.addMutationHandler(custom_function2, 2, ...) >>> ga.setSelectionHandler(custom_function3)
```

2.5.4 1.4 Adding fitness function of the GA

Users can define their custom fitness function and add it to the GA.

A typical fitness signature would be *def fitness(chromosome)*

```
>>> def fitness(chromosome):
>>>     return sum(chromosome)
```

It can be added like `>>> ga.setFitnessHandler(fitness)`

For fitness functions which depend on more than just the chromosome(eg: in TSP), we can add more parameters like *def fitness(chromosome, ...)*

Eg: In TSP

```
>>> def TSP(chromosome, matrix):
>>>     total = 0
>>>     for i in range(len(chromosome)-1):
>>>         total += matrix[chromosome[i]][chromosome[i+1]]
>>>     return total
>>> ga.setFitnessHandler(fitness, TSP_matrix)
```

2.5.5 1.5 Running the GA

The GA can now be executed `>>> ga.evolve(20)` where the parameter to be given is the number of GA iterations to be executed

In case, you feel you want to continue from a previous execution, you can `>>> ga.continue_evolve(20)`

We can obtain the best member after the evolution by

```
>>> print(ga.best_fitness)
```

This returns a tuple where the first element is the best chromosome and the second element is the corresponding best fitness value

2.5.6 1.6 Statistics

By default, we can view the following GA statistics after Evolution - `'best-fitness'`, `'worst-fitness'`, `'avg-fitness'`, `'diversity'`, `'mutation_rate'`

We can plot graphs for this

```
>>> import matplotlib.pyplot as plt
>>> fig = ga.statistics.plot_statistics(['best-fitness', 'worst-fitness', 'avg-fitness',
↵'])
>>> plt.show()
```

We can also define custom Statistics and add it to the GA

```
>>> def range_of_generation(fitness_mappings, ga):
>>>     return abs(fitness_mappings[0][1] - fitness_mappings[-1][1])
```

```
>>> ga.addStatistic('range', range_of_generation)
# After evolutions
>>> fig = ga.statistics.plot_statistics(['range'])
>>> plt.show()
```

2.5.7 1.7 Custom Evolutions

Users can define some custom evolution by subclassing `BaseEvolution` and filling `ga.population.new_members` with the new members from the evolution in the `def evolve(self, ga)` function where `ga` is the `GAEngine` object. Return 1 from this function if the required fitness value is found else no need to return anything

```
>>> from pygenetic import Evolution
>>> class CustomEvolution(Evolution.BaseEvolution):
>>>     def __init__(self, ...):
>>>         ....
```

```

>>> def evolve(self, ga):
    # Carry out custom evolution
    # Current population is at ga.population.members
    ### Note:
    ### ga.handle_selection() does the selection using the given selection handler
    ### Fitness mappings are present at ga.fitness_mappings
    ### ga.chooseCrossoverHandler() chooses
    ### ga.doCrossover(crossoverHandler,father,mother) executes crossover
    ### ga.chooseMutationHandler() chooses
    ### ga.doMutation(mutationHandler,chromosome) does mutation
    # Fill ga.population.new_members with the new population from evolution
    # Return 1 if the required fitness value is found else no need to return anything

```

You can then create the custom Evolution Instance and add it to the GAEngine

```

>>> evolution = CustomEvolution(...)
>>> ga.setEvolution(evolution)

```

2.6 2. Usage of *SimpleGA*: the High Level pygenetic GA API

Very Simple GAs can be executed using *SimpleGA*

```

>>> from pygenetic import SimpleGA
>>> ga = SimpleGA.SimpleGA(minValue=1,maxValue=120,
                           noOfGenes=20,fitness_func=lambda x:sum(x),
                           duplicates=False,population_size=1000,
                           fitness_type='max')

```

where *minValue* is the minimum value a gene can take *maxValue* is the maximum value a gene can take *noOfGenes* is the number of genes in a chromosome *duplicates* determines if duplicates in the chromosome are allowed *fitness_func* is the fitness function *population_size* is the size of the population *fitness_type* is the fitness type (similar to *GAEngine*)

Other parameters include * *cross_prob*: crossover probability * *mut_prob*: mutation probability * *crossover_handler*: can one of the the following values 'distinct', 'onePoint', 'twoPoint', 'PMX' and 'OX' (default='onePoint') * *mutation_handler*: can take one of the following values 'swap' and 'bitFlip' (default='swap') * *selection_handler*: can take one of the following values 'best', 'rank' and 'roulette' (default='swap')

It can then be run

```

>>> ga.evolve(10)

```

for evolving it for 10 generations

2.6.1 3. Best ANN Topology Finder

Users can find best ANN Topology to train for a classification problem using GA

```

>>> from pygenetic import ANNEvolve
>>> import numpy
# load pima indians dataset
>>> dataset = numpy.loadtxt("input.csv", delimiter=",")
# split into input (X) and output (Y) variables

```

(continues on next page)

(continued from previous page)

```
>>> X = dataset[:,0:8]
>>> Y = dataset[:,8]
>>> a = ANNEvolve.ANNTopologyEvolve(X,Y,hiddenLayers=2,population_size=10,
                                     neuronsPerLayer=[2,5,10,12],activations=['relu','sigmoid'],
                                     optimizers=['adam'],loss='binary_crossentropy',
                                     metrics='accuracy',epochs=30,batch_size=10)
>>> a.evolve(100)
>>> print(a.best_fitness)
```


CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`