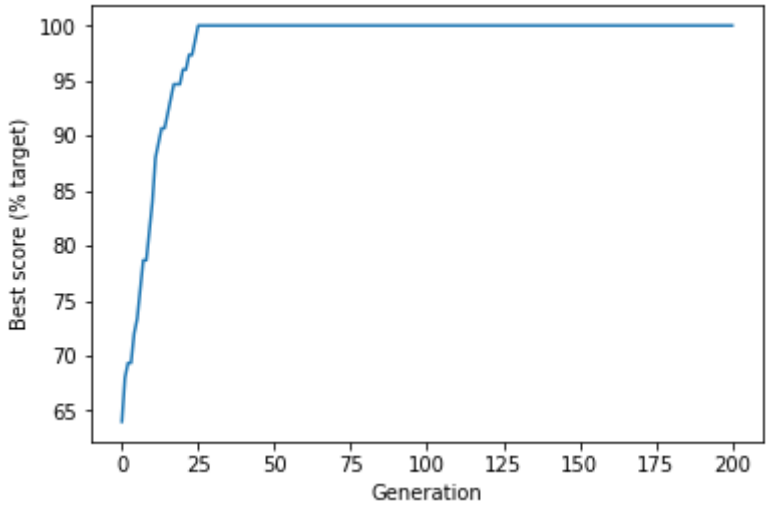
94: Genetic algorithms 1. A simple genetic algorithm

Michael Allen . Algorithms October 1, 2018September 22, 2019 8 Minutes

Note: For core code only, without explanation or test code sections see this link: code_only (https://pythonhealthcare.org/2018/10/01/94-genetic-algorithms-a-simple-genetic-algorithm-code-only/)



For more discussion on the general concepts of genetic algorithms, which are only presented briefly here (as we will focus on how to code a simple example in Python), see Wikipedia article (https://en.wikipedia.org/wiki/Genetic_algorithm).

In this example we will look at a basic genetic algorithm (GA). We will set up the GA to try to match a pre-defined 'optimal. solution. Often with GAs we are using them to find solutions to problems which 1) cannot be solved with 'exact' methods (methods are are guaranteed to find the best solution), and 2) where we cannot recognise when we have found the optimal solution. GAs therefore fall into a collection of algorithms called heuristic (from Greek for 'search') algorithms that hunt down good solutions, without us knowing how far off the theoretical optimal solution they are.

In this case however we will test the algorithms against a solution we know we are looking for.

Principles of genetic algorithms (GAs)

GAs are iterating algorithms, that is they repeatedly loop through a progress until a target is reached or a maximum number of iterations (called 'generations' in GAs) is reached.

The basic steps of a genetic algorithm are:

1) Create a population of randomly generated solutions, coded as binary arrays, and score population for performance (or 'fitness') of each individual.

2) Loop (until target performance is reached or a maximum number of generations is reached):

- Select two parents to 'breed'. Selection is based on performance (or 'fitness') – better performing parents are more likely to be selected.
- Generate two child solutions from two parents by mixing the genes from each parent and by applying a chance of random mutation.
- Repeat child generation until a required new population size is reached.
- Score new population

The solution to 'hunt down'

We code most GAs to work with binary strings (or a binary NumPy array, as we will use here), so that the solution may be represented as a series of 0 and 1.

A real-life example in healthcare is that the array of 0s and 1s may represents the choices of closed or open hospital units providing a given service. We then evaluate each solution against predetermined criteria.

Here we will define a known solution based on a string of 70 0s or 1s. The number of possible combinations for this is 2^18, or 1.2 x 10^18 – that is 1 followed by eighteen zeros. Or, to put it another way (as these large numbers are difficult to imagine) …… the universe is about 15 billion (15 x 10^9) years old, or 5 x 10^17 seconds old. If we could evaluate 1,000 solutions per second, then a computer would need to run for twice the current age of the universe in order to evaluate all possible combinations. Let's see how close to the perfect solution we can get in reasonable time!

In GA we will call each potential solution to be evaluated a 'chromsome'. Each element (a 0 or 1) in that chromsome is a 'gene'.

```
import random
import numpy as np

def create_reference_solution(chromosome_length):

    number_of_ones = int(chromosome_length / 2)

    # Build an array with an equal mix of zero and ones
    reference = np.zeros(chromosome_length)
    reference[0: number_of_ones] = 1

    # Shuffle the array to mix the zeros and ones
    np.random.shuffle(reference)

    return reference
```

Let's test the function and show an example 70-gene reference chromosome.

```
# Print an example target array
print (create_reference_solution(70))
```

```
OUT:

[1. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 1. 1. 0.
 1. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1.
 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 1.]
```

Create a starting random population

We will use NumPy to store our population. Each row will represent one solution, which will contain a random sequence of zeros and ones. The number of rows will represent to number of 'individuals, in our population.

```
def create_starting_population(individuals, chromosome_length):
    # Set up an initial array of all zeros
    population = np.zeros((individuals, chromosome_length))
    # Loop through each row (individual)
    for i in range(individuals):
        # Choose a random number of ones to create
        ones = random.randint(0, chromosome_length)
        # Change the required number of zeros to ones
        population[i, 0:ones] = 1
        # Sfuffle row
        np.random.shuffle(population[i])

    return population
```

Let's test by showing a random population of 4 individuals with a gene length of 10.

```
print (create_starting_population(4, 10))
```

```
OUT:

[[0. 1. 1. 1. 1. 0. 0. 1. 1. 0.]
 [1. 1. 1. 1. 0. 1. 1. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 0. 1. 1. 0.]
 [1. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
```

Calculate fitness of population

In GAs we refer to how good each individual in the population is, as 'fitness'. The calculate_fitness function will be the evaluation procedure you wish to apply in your algorithm. In this example we are going to return the number of genes (elements) in a potential solution (chromosome) that match our f=reference standard.

```
def calculate_fitness(reference, population):
    # Create an array of True/False compared to reference
    identical_to_reference = population == reference
    # Sum number of genes that are identical to the reference
    fitness_scores = identical_to_reference.sum(axis=1)

    return fitness_scores
```

Let's test what we have so far:

```
reference = create_reference_solution(10)
print ('Reference solution: \n', reference)
population = create_starting_population(6, 10)
print ('\nStarting population: \n', population)
scores = calculate_fitness(reference, population)
print('\nScores: \n', scores)
```

```
OUT:

Reference solution:
 [1. 1. 0. 0. 0. 0. 1. 0. 1. 1.]

Starting population:
 [[0. 0. 1. 1. 1. 0. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1. 1. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [1. 1. 1. 1. 0. 0. 1. 1. 0. 1.]
 [1. 1. 0. 0. 0. 0. 0. 1. 1. 1.]]

Scores:
 [3 5 6 6 6 8]
```

Choosing individuals to breed with tournament selection

Genetic algorithms mimic biology in that the individuals with the best fitness cores are most likely to breed and pass on their genes. But we do not simply take all the best individuals from our population to breed, as this might risk 'in-breeding'. Rather, we use a method that means better individuals are moire likely to breed, but low fitness individuals at times may be chosen to breed.

In tournament selection we first choose two individuals at random from our population (it is possible that two low fitness individuals may be chosen). We then pass those individuals to a 'tournament' where the individual with the highest fitness will be chosen.

It is possible to further modify this so that the highest fitness individual will win with a given probability, but we will keep it simple here and have the highest fitness individual always winning. It is also possible to have more than two individuals in a tournament. The more individuals in a tournament the more the picked population will be biased towards the highest fitness individuals.

```
def select_individual_by_tournament(population, scores):
    # Get population size
    population_size = len(scores)

    # Pick individuals for tournament
    fighter_1 = random.randint(0, population_size-1)
    fighter_2 = random.randint(0, population_size-1)

    # Get fitness score for each
    fighter_1_fitness = scores[fighter_1]
    fighter_2_fitness = scores[fighter_2]

    # Identify undividual with highest fitness
    # Fighter 1 will win if score are equal
    if fighter_1_fitness >= fighter_2_fitness:
        winner = fighter_1
    else:
        winner = fighter_2

    # Return the chromsome of the winner
    return population[winner, :]
```

Let's test selection of parents:

```
# Set up and score population
reference = create_reference_solution(10)
population = create_starting_population(6, 10)
scores = calculate_fitness(reference, population)

# Pick two parents and dispplay
parent_1 = select_individual_by_tournament(population, scores)
parent_2 = select_individual_by_tournament(population, scores)
print (parent_1)
print (parent_2)
```

```
OUT:

[1. 0. 1. 1. 0. 1. 0. 1. 1. 0.]
[1. 0. 1. 1. 0. 1. 0. 1. 1. 0.]
```

Producing children from parents – crossover

When two individuals are chosen, the next step is to produce 'children' from them. We produce these children by 'crossover' mix of their two chromosomes. We choose a random point within the chromosome, and then one 'child' will take the left portion (up to, but not including, the crossover point) from parent 1 and the corresponding right portion from parent 2. The result is a mix of genes from each parent. The second 'child' will be the opposite of this – portion (up to, but not including) the crossover point) from parent 2 and the corresponding right portion from parent 1.

It is possible to have more than one crossover point, but we will keep it simple and have a single crossover point.

In [9]:

```
def breed_by_crossover(parent_1, parent_2):
    # Get length of chromosome
    chromosome_length = len(parent_1)

    # Pick crossover point, avoding ends of chromsome
    crossover_point = random.randint(1,chromosome_length-1)

    # Create children. np.hstack joins two arrays
    child_1 = np.hstack((parent_1[0:crossover_point],
                         parent_2[crossover_point:]))

    child_2 = np.hstack((parent_2[0:crossover_point],
                         parent_1[crossover_point:]))

    # Return children
    return child_1, child_2
```

And let's test it so far, creating a population, scoring it, picking two 'parents' and producing 'two children'.

```
# Set up and score population
reference = create_reference_solution(15)
population = create_starting_population(100, 15)
scores = calculate_fitness(reference, population)

# Pick two parents and dispplay
parent_1 = select_individual_by_tournament(population, scores)
parent_2 = select_individual_by_tournament(population, scores)

# Get children
child_1, child_2 = breed_by_crossover(parent_1, parent_2)

# Show output
print ('Parents')
print (parent_1)
print (parent_2)
print ('Children')
print (child_1)
print (child_2)
```

```
OUT:

Parents
[1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 1. 1. 0.]
Children
[1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

Random mutation of genes

In evolution sometimes genes are copied incorrectly. This change may be harmful or beneficial. We mimic this by having a certain probability of that a gene (which is either a 0 or a 1) becomes switched.

Typically this probability is low (e.g. 0.005), though it can be made to be flexible (e.g. increase mutation rate if progress has stalled)

```
def randomly_mutate_population(population, mutation_probability):

    # Apply random mutation
        random_mutation_array = np.random.random(
            size=(population.shape))

        random_mutation_boolean = \
            random_mutation_array <= mutation_probability

        population[random_mutation_boolean] = \
        np.logical_not(population[random_mutation_boolean])

        # Return mutation population
        return population
```

Let's test our function with a high mutation rate (0.25) to see the effects. You can change the mutation rate and see what happens (a mutation rate of 1.0 will invert all genes).

```
# Set up and score population
reference = create_reference_solution(15)
population = create_starting_population(100, 15)
scores = calculate_fitness(reference, population)

# Pick two parents and display
parent_1 = select_individual_by_tournament(population, scores)
parent_2 = select_individual_by_tournament(population, scores)

# Get children and make new population
child_1, child_2 = breed_by_crossover(parent_1, parent_2)
population = np.stack((child_1, child_2))

# Mutate population
mutation_probability = 0.25
print ("Population before mutation")
print (population)
population = randomly_mutate_population(population, mutation_probability)
print ("Population after mutation")
print (population)
```

```
OUT:

Population before mutation
[[1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 1. 1.]
 [0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1.]]

Population after mutation
[[1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1.]
 [0. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 1.]]
```

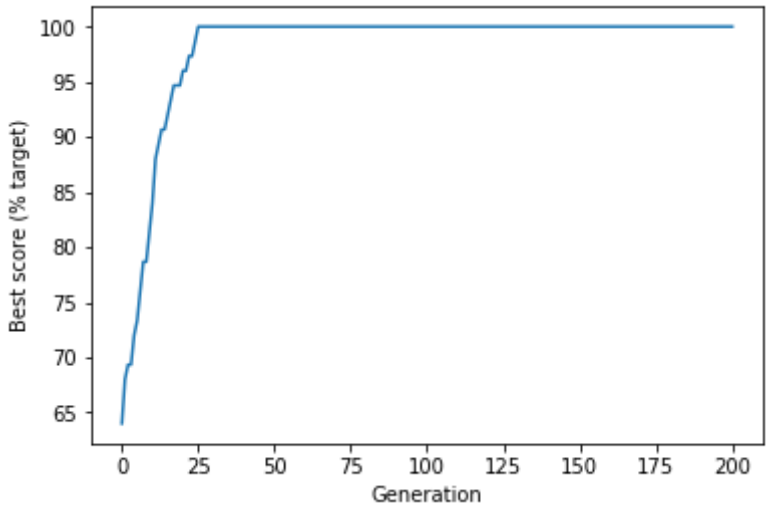
Putting it all together

We've defined all the functions we need. Now let's put it all together.

```
# Set general parameters
chromosome_length = 75
population_size = 500
maximum_generation = 200
best_score_progress = [] # Tracks progress

# Create reference solution
# (this is used just to illustrate GAs)
reference = create_reference_solution(chromosome_length)

# Create starting population
population = create_starting_population(population_size, chromosome_length)

# Display best score in starting population
scores = calculate_fitness(reference, population)
best_score = np.max(scores)/chromosome_length * 100
print ('Starting best score, percent target: %.1f' %best_score)

# Add starting best score to progress tracker
best_score_progress.append(best_score)

# Now we'll go through the generations of genetic algorithm
for generation in range(maximum_generation):
    # Create an empty list for new population
    new_population = []

    # Create new popualtion generating two children at a time
    for i in range(int(population_size/2)):
        parent_1 = select_individual_by_tournament(population, scores)
        parent_2 = select_individual_by_tournament(population, scores)
        child_1, child_2 = breed_by_crossover(parent_1, parent_2)
        new_population.append(child_1)
        new_population.append(child_2)

    # Replace the old population with the new one
    population = np.array(new_population)

    # Apply mutation
    mutation_rate = 0.002
    population = randomly_mutate_population(population, mutation_rate)

    # Score best solution, and add to tracker
    scores = calculate_fitness(reference, population)
    best_score = np.max(scores)/chromosome_length * 100
    best_score_progress.append(best_score)

# GA has completed required generation
print ('End best score, percent target: %.1f' %best_score)

# Plot progress
import matplotlib.pyplot as plt
```

```
%matplotlib inline
plt.plot(best_score_progress)
plt.xlabel('Generation')
plt.ylabel('Best score (% target)')
plt.show()
```

```
OUT:

Starting best score, percent target: 64.0
End best score, percent target: 100.0
```



Tagged:
genetic algorithm,
health service research,
healthcare modelling,
python

Published by Michael Allen

Interests are use of simulation and machine learning in healthcare, currently working for the NHS and the University of Exeter. Committed to all work being performed in Free and Open Source Software (FOSS), and as much source data being made available as possible. https://gitlab.com/michaelallen1966 View all posts by Michael Allen

12 thoughts on “94: Genetic algorithms 1. A simple genetic algorithm”

Pingback: Index – Python for healthcare analytics and modelling
Pingback: 94. Genetic Algorithms 1. A simple genetic algorithm (code only) – Python for healthcare analytics and modelling
Pingback: 95: When too many multi-objective solutions exist: selecting solutions based on crowding distances – Python for healthcare analytics and modelling
Pingback: 117. Genetic Algorithms 2 – a multiple objective genetic algorithm (NSGA-II) – Python for healthcare modelling and data science
notjustanotherguy says:
January 8, 2020 at 3:08 pm
Thanks very much for this! Just a little question: Why do you divide by two in the following snippet:

# Create new popualtion generating two children at a time
for i in range(int(population_size/2)):

◎ Reply

Michael Allen says:
January 8, 2020 at 3:52 pm
It is because for each iteration of the loop you take two parents and generate two children. So without the /2 you would double the population size.

◎ Reply

Emirhan U says:
January 12, 2020 at 10:36 am
It might choose the same chromosome as a second parent. And seems like it did in your code.

◎ Reply

Michael Allen says:
January 12, 2020 at 9:02 pm
Yes, but that’s not a problem. In fact I commonly use an adaptation where the best parents are also kept.

◎ Reply

rookieninja says:
February 5, 2020 at 3:08 am
How can I use GA with Neural Networks? Also, Is there any notable libraries in python to use GA??

◎ Reply

Michael Allen says:
February 5, 2020 at 8:28 am
Hello. What are trying to achieve with GAs and neural networks combined. I’m afraid I can’t vouch for any libraries for GAs in Python as we have always coded our own. People have recommended DEAP to me, but I can’t really say more than that.

◎ Reply

sergey shubin says:
February 27, 2020 at 7:44 pm
and also I forget X, Z have boundaries from 0 to 640, and Y from 0 to 10000. Both minimised functions, which I described above have constraint not to be negative, it means results have to be >=0
sergeyshubin2013@gmail.com

◎ Reply

Michael Allen says:
February 28, 2020 at 7:35 am
Hello. I the problem you described (I'm not sure why the first comment is not showing sorry), each parameter would be coded as a gene in the chromosome (initialise values using random uniform in a range you think reasonable). The only other difference is that for the mutation rather than switching between 0 and 1 you will want to multiply or divide by a factor (e.g. pick a random number between 1 and 1.5).

For boundaries you have three options:

* Remove all individuals that go outside of the boundaries
* Censor the values such that values outside of the boundaries (e.g. caused by mutation) are reset to the boundary limits
* Have the objective function penalise each boundary transgression

Have you been told you need a genetic algorithm to solve this? It looked a little like a problem that could be solved with linear programming.

If your aim is to learn GAs then I would write it yourself. Otherwise I might use a GA library like DEAP.

Hope that helps

◎ Reply