**Genetic Algorithms Introduction**

https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

Good basic explanation above

## Creating a genetic algorithm for beginners

### Introduction

A genetic algorithm (GA) is great for finding solutions to complex search problems. They're often used in fields such as engineering to create incredibly high quality products thanks to their ability to search a through a huge combination of parameters to find the best match. For example, they can search through different combinations of materials and designs to find the perfect combination of both which could result in a stronger, lighter and overall, better final product. They can also be used to design computer algorithms, to schedule tasks, and to solve other optimization problems. Genetic algorithms are based on the process of evolution by natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Surprisingly although genetic algorithms can be used to find solutions to incredibly complicated problems, they are themselves pretty simple to use and understand.

### How they work

As we now know they're based on the process of natural selection, this means they take the fundamental properties of natural selection and apply them to whatever problem it is we're trying to solve.

The basic process for a genetic algorithm is:

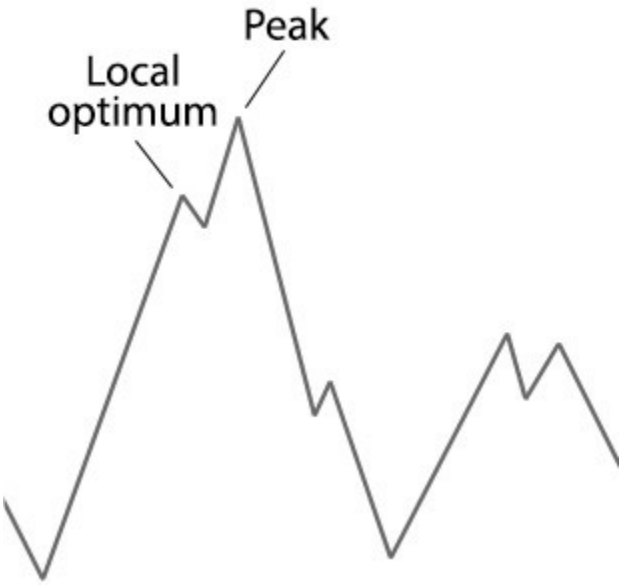
1. Initialization - Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.
2. Evaluation - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.
3. Selection - We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.
4. Crossover - During crossover we create new individuals by combining aspects of our selected individuals. We can think of this as mimicking how sex works in nature. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from each of it's parents.
5. Mutation - We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individuals genome.
6. And repeat! - Now we have our next generation we can start again from step two until we reach a termination condition.

### Termination

There are a few reasons why you would want to terminate your genetic algorithm from continuing it's search for a solution. The most likely reason is that your algorithm has found a solution which is good enough and meets a predefined minimum criteria. Offer reasons for terminating could be constraints such as time or money.

### Limitations

Imagine you were told to wear a blindfold then you were placed at the bottom of a hill with the instruction to find your way to the peak. You're only option is to set off climbing the hill until you notice you're no longer ascending anymore. At this point you might declare you've found the peak, but how would you know? In this situation because of your blindfolded you couldn't see if you're actually at the peak or just at the peak of smaller section of the hill. We call this a local optimum. Below is an example of how this local optimum might look:




Unlike in our blindfolded hill climber, genetic algorithms can often escape from these local optimums if they are shallow enough. Although like our example we are often never able to guarantee that our genetic algorithm has found the global optimum solution to our problem. For more complex problems it is usually an unreasonable exception to find a global optimum, the best we can do is hope for is a close approximation of the optimal solution.

Now we're going to put together a simple example of using a genetic algorithm in Python. We're going to optimize a very simple problem: trying to create a list of `N` numbers that equal `X` when summed together.

If we set `N = 5` and `X = 200`, then these would all be appropriate solutions.

```
lst = [40,40,40,40,40]
lst = [50,50,50,25,25]
lst = [200,0,0,0,0]
```

## Ingredients of The Solution

Each suggested solution for a genetic algorithm is referred to as an **individual**. In our current problem, each list of `N` numbers is an individual.

```
>>> from random import randint
>>> def individual(length, min, max):
...     'Create a member of the population.'
...     return [ randint(min,max) for x in xrange(length) ]
...
>>> individual(5,0,100)
[79, 0, 20, 47, 40]
>>> individual(5,0,100)
[64, 1, 25, 84, 87]
```

The collection of all *individuals* is referred to as our **population**.

```
>>> def population(count, length, min, max):
...     """
...     Create a number of individuals (i.e. a population).
...
...     count: the number of individuals in the population
...     length: the number of values per individual
...     min: the min possible value in an individual's list of values
...     max: the max possible value in an individual's list of values
...
...     """
...     return [ individual(length, min, max) for x in xrange(count) ]
...
>>> population(3,5,0,100)
[[51, 55, 73, 0, 80], [3, 47, 18, 65, 55], [17, 64, 77, 43, 48]]
```

Next we need a way to judge the how effective each solution is; to judge the **fitness** of each *individual*. Predictably enough, we call this the **fitness function**. For our problem, we want the fitness to be a function of the distance between the sum of an individuals numbers and the target number `X`.

We can implement the *fitness function* as follows:

```
>>> from operator import add
>>> def fitness(individual, target):
...     """
...     Determine the fitness of an individual. Lower is better.
...
...     individual: the individual to evaluate
...     target: the sum of numbers that individuals are aiming for
...     """
...     sum = reduce(add, individual, 0)
...     return abs(target-sum)
...
>>> x = individual(5,0,100)
>>> fitness(x, 200)
165
```

Personally, I'd prefer to have a high fitness score correlate to a fit individual rather than the current implementation where a perfectly fit individual has a fitness of `0`, and the higher the worse. Ah well, regardless, keep that detail in mind while following this code.

It's also helpful to create a function that will determine a *population's* average *fitness*.

```
>>> def grade(pop, target):
...     'Find average fitness for a population.'
...     summed = reduce(add, (fitness(x, target) for x in pop), 0)
...     return summed / (len(pop) * 1.0)
...
>>> x = population(3,5,0,100)
```

```
>>> target = 200
>>> grade(x, target)
116
```

Now we just need a way evolve our population; to advance the*population* from one **generation** to the next.

## Evolution

This is the secret sauce of genetic algorithms, where secret means fairly obvious, and sauce means sauce. Consider a population of elk which are ruthlessly hunted by a pack of wolves. With each generation the weakest are eaten by the wolves, and then the strongest elk reproduce and have children. Abstract those ideas a bit, and we can implement the evolution mechanism.

1. For each generation we'll take a portion of the best performing individuals as judged by our *fitness function*. These high-performers will be the parents of the next generation.

   We'll also randomly select some lesser performing individuals to be parents, because we want to promote genetic diversity. Abandoning the metaphor, one of the dangers of optimization algorithms is getting stuck at a local maximum and consequently being unable to find the real maximum. By including some individuals who are not performing as well, we decrease our likelihood of getting stuck.

2. Breed together parents to repopulate the population to its desired size (if you take the top 20 individuals in a population of 100, then you'd need to create 80 new children via breeding). In our case, breeding is pretty basic: take the first `N/2` digits from the father and the last `N/2`digits from the mother.

```
>>> father = [1,2,3,4,5,6]
>>> mother = [10,20,30,40,50,60]
>>> child = father[:3] + mother[3:]
>>> child
[1,2,3,40,50,60]
```

   It's okay to have one parent breed multiple times, but one parent should never be both the father and mother of a child.

8. Merge together the parents and children to constitute the next generation's population.

9. Finally we **mutate** a small random portion of the population. What this means is to have a probability of randomly modifying each individual.

```
>>> from random import random, randint
>>> chance_to_mutate = 0.01
>>> for i in population:
...     if chance_to_mutate > random():
...         place_to_modify = randint(0,len(i))
...         i[place_to_modify] = randint(min(i), max(i))
...
```

   This--just like taking individuals who are not performing particularly well--is to encourage genetic diversity, i.e. avoid getting stuck at local maxima.

Putting it all together, the code to evolve a generation can be implemented like this:

```
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [ (fitness(x, target), x) for x in pop]
```

```
    graded = [ x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]

    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))

    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:half] + female[half:]
            children.append(child)

    parents.extend(children)
    return parents
```

Now we've written all the pieces of a genetic algorithm, and we just have to try it out and see if it works.

## Testing It Out

Here is a simple way to use the code we've written:

```
>>> target = 371
>>> p_count = 100
>>> i_length = 5
>>> i_min = 0
>>> i_max = 100
>>> p = population(p_count, i_length, i_min, i_max)
>>> fitness_history = [grade(p, target),]
>>> for i in xrange(100):
...     p = evolve(p, target)
...     fitness_history.append(grade(p, target))
...
>>> for datum in fitness_history:
...    print datum
...
```

Running that code, you'll get to watch as generations' fitness gradually (but non-deterministically) approach zero. The output of one of my runs looked like this:

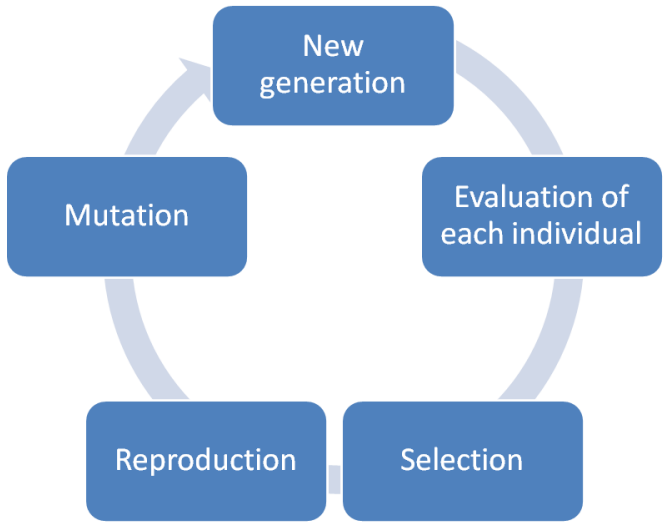
```
['76.06', '32.13', '26.34', '18.32', '15.08', '11.69', '14.05', '9.460', '4.950', '0.0', '0.0', '0
.0', '0.0', '0.0', '0.800', '0.0', '0.239', '0.780', '0.0', '0.0', '0.0', '0.0', '1.48', '0.0', '0
.0', '0.0', '0.0', '0.0', '0.0', '0.149', '0.239', '0.12', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0
', '0.0', '0.149', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '4.200', '0.0', '2.049', '0.0', '0.20
0', '0.080', '0.0', '1.360', '0.0', '0.0', '0.0', '0.0', '1.399', '0.0', '0.0', '0.149', '1.389',
'1.24', '0.0', '0.16', '0.0', '0.680', '0.0', '0.0', '1.78', '1.05', '0.0', '0.0', '0.0', '0.0', '
1.860', '4.080', '3.009', '0.140', '0.0', '0.38', '0.0', '0.0', '0.0', '0.0', '0.0', '2.189', '0.0
', '0.0', '3.200', '1.919', '0.0', '0.0', '4.950', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0']
```

With 20% survival (plus an additional 5% of other individuals) and 1% mutation, it only took nine generations to reach a perfect solution. Then the algorithm joyfully runs in circles for as long as you'll let the mutations continue But this is a good feeling right? If it only took us half an hour to solve a problem of this magnitude, imagine what we could do with a day. A genetic algorithm for optimizing your Apache2 configuration file for number of children processes?

## How to create a good Artificial Intelligence?

The naive solution is to create an "empirical algorithm" which is a set of rules: "if you meet this conditions, act like that". I could imagine that with enough rules like this we could reproduce natural intelligence. But the amount of work is colossal and the final solution will never be able to best its creator. Isn't it depressing to spend a lot of time creating something while knowing it can't be perfect? To avoid that, a new idea emerged. What if instead of creating a direct solution we recreate evolution. We could create the first prehistoric fishes, put them in conditions suitable to evolution and let them evolve freely toward man-kind or even further. This idea is called "genetic algorithm" and we are going to build one in the next part. So first let us refresh our memories and try to understand the natural selection theorized by Darwin.

This theory is simple: if a population want to thrive, it must improve by itself constantly, it's the survival of the fittest. The best element of the population should inspire the offspring, but the other individuals must not be forgotten in order to maintain some diversity and be able to adapt in case of a variation of the natural environment.

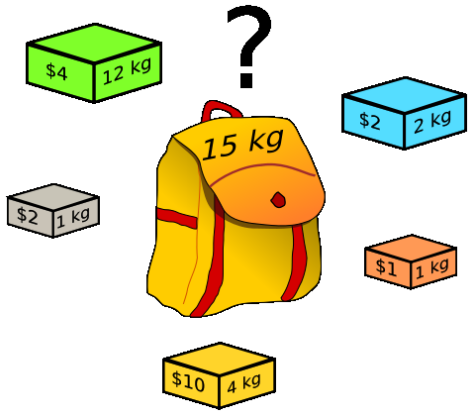



How genetic algorithms work, from one generation to the other

Genetic algorithms are especially efficient with **optimization** problems.

## Example: the Knapsack problem

The backpack optimization is a classical algorithm problem. You have two things: a backpack with a size (the weight it can hold) and a set of boxes with different weights and different values. The goal is to fill the bag to make it as valuable as possible without exceeding the maximum weight. It is a famous mathematical problem since 1972. *The genetic algorithm is well suited to solve that because it's an optimization problem with a lot of possible solutions.*




The Knapsack problem can be solved efficiently with a genetic algorithm

---

In order to test by ourselves how a genetic algorithm works, we are going to use it to solve a simple problem: how could I crack my colleague's password?
The algorithm is implemented on Python 3.

## Choosing a fitness function

**The evaluation function is the first step to create a genetic algorithm**. It's the function that estimates the success of our specimen: it will allow us to divide the population between the ugly duckling and the beautiful swans. The goal of this separation is that, later, the successful specimen will have more "chance" to get picked to form the next generation. As simple as it appears, don't be fooled, it's the step of a genetic algorithm with the more *intelligence* related to the problem.
What is our goal? Crack a password. Thus the goal of our function is to transform the binary result "fail or success" in a continuous mark from 0 (can't fail more) to 100 (perfection).

The simplest solution here is:

```
fitness score = (number of char correct) / (total number of char)
```

That way, an individual with a bigger fitness result is a specimen genetically closer to success than the others. Thus the fitness function for our genetic algorithm will accurately sort the population.
The fitness function of our genetic algorithm

## Creating our individuals

So now we know how to evaluate our individuals; but how do we define them? This part is really tricky: the goal is to know what are the unalterable characteristics and what is variable.
The comparison with genetics is here really helpful. Indeed, the DNA is composed of genes, and each of those genes comes through different alleles (different versions of this gene). Genetic algorithms retain this concept of population's DNA.
In our case, our individuals are going to be words (obviously of equal length with the password). Each letter is a gene and the value of the letter is the allele. In the word "banana": 'b' is the allele of the first letter.
What is the point of this creation?

- We know that each of our individuals is keeping the good shape (a word with the correct size)

- Our population can cover every possibility (every word possible with this size).

Out genetic algorithm can then explore all possible combinations.

## Creating our first population

Now, we know what are the characteristics of our individuals and how we can evaluate their performance. We can now start the "evolution" step of our genetic algorithm.

The main idea to keep in mind when we create the first population is that we must not point the population towards a solution that seems good. We must make the population as wide as possible and make it cover as many possibilities as possible. The perfect first population of a genetic algorithm should cover every existing allele.

So in our case, we are just going to create words only composed of random letters.

The evolution step of our genetic algorithm

## From one generation to the next

Given a generation, in order to create the next one, we have 2 things to do. First we select a specific part of our current generation. Then the genetic algorithm combines those breeders in order to create the next batch.

### *Breeders selection*

They are lots of way to do this but you must keep in mind two ideas: the goals are to select the best solutions of the previous generation and not to completely put aside the others. The hazard is: if you select only the good solutions at the beginning of the genetic algorithm you are going to converge really quickly towards a local minimum and not towards the best solution possible.

My solution to do that is to select on the one hand the N better specimen (in my code, N = best_sample) and on the other hand to select M random individuals without distinction of fitness (M = lucky_few).

The breeders selection in our genetic algorithm

### *Breeding*

We can keep on the biologic analogy for the breeding in our genetic algorithm. The goal of sexual reproduction is to mix the DNA of 2 individuals, so let's do the same thing here. We have two individuals: *"Tom"* and "*Jerry*", their DNA is defined by their alleles (the value of each letter). Thus in order to mix their DNA, we just have to mix their letters. There are lots of ways to do this so I picked the simplest solution: for each letter of the child, take randomly the letter of "*Tom*" or "*Jerry*".

*NB: Obviously, the couple "*Tom*" and "*Jerry*" isn't going to produce only one child. You have to fix the number of children per couple in order to keep a stable population in your genetic algorithm. The number of individuals in the generation 0 equals the number of individuals in the next generation.*

The breeding step of our genetic algorithm

## Mutation

This last step of our genetic algorithm is the natural mutation of an individual. After the breeding, each individual must have a small probability to see their DNA change a little bit. The goal of this operation is to prevent the algorithm to be blocked in a local minimum.

Content is from these articles: CREDITS

https://lethain.com/genetic-algorithms-cool-name-damn-simple/
https://blog.sicara.com/getting-started-genetic-algorithms-python-tutorial-81ffa1dd72f9
http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3
https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
https://www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/

https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6