**Documentation about GENETIC ALGORITHM written by me**

(*This is an article by me in which I have written about a subpart of Artificial intelligence and Machine learning that I read and implemented recently, called GENETIC ALGORITHM.)*

“Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand”

Where else can we learn how to solve problems more effectively than from nature? Nature has always been a fantastic source of a wide range of phenomena and items that leave people perplexed as to how they ever came into being. The natural world itself can actually provide answers to a lot of questions. Living organisms are exemplary problem solvers. They exhibit a versatility that puts the best computer programs to shame. Computer scientists find this fact especially infuriating because they may put months or years of thought into developing an algorithm, yet creatures acquire their skills through the apparently random process of **evolution** and **natural selection.**

Natural selection is a very integral concept of evolution. If examined and utilized in AI then it eliminates one of the greatest hurdles in software design which is ‘specifying in advance the features of a program the actions it should take to deal with them’. Thus, by using the methods of evolution, one can ‘breed’ the codes that can solve problems that are complex and whose structure is yet not fully understood.

These methods which utilize the evolutionary concepts of natural selection such as the “mixing of genes through sexual reproduction” and “Survival of the Fittest” are called GENETIC ALGORITHMS. It uses the Darwinian principle of natural selection using patterns such as mutation and crossover to come up with a solution.

Genetic Algorithms rely on two main strategies to find the optimal solutions: **Mutation** and **Crossover.**

**Using Crossover to Exploit**: Crossover is how genetic algorithms exploit in search. Crossover is the process of creating new child solutions from parent solutions. The idea is that the strongest solutions have characteristics that make them strong. These characteristics are called schemas. Schemas are building blocks of fit solutions.

**Using Mutation to Explore:** Mutation is how genetic algorithms explore. It’s not enough to simply keep trying to build new solutions from previous ones, which is essentially the same as trying the same path over and over again. Mutation introduces randomness into the genetic algorithms. The goal is to slightly alter some aspects of the previous solutions to create newer solutions, which may lead to newer, better paths.

Now having some idea about what GAs are, the following is a basic overview of how GAs work:

1. Individuals in a population compete for resources.
2. Those that are successful (fittest) mate to create more offspring.
3. Each inherent quality of the parent can be thought of as its gene.
4. Genes from the “fittest” parents are passed on to the offspring, which is sometimes better than the parent.
5. Hence, the offspring is better suited for the environment(more optimized).

Since GAs operate on survival of the fittest, we give a score based on how good the individual is at surviving. Fitness can be a function of the initial parameters and used in successive genetic operations. Individuals with higher fitness have a higher chance to reproduce and pass on their genes. Hence, that’s why the population is usually sorted based on its fitness.

The above-stated algorithmic steps need to be converted into a sort of code and then implemented in the computers so that they can work:

* **Initialize** a population with random candidates
* **Evaluate** all individuals
* **While** the termination criteria not met
* **Select** parents from those individuals
* **Apply** the crossover between the parents to produce the offsprings
* **Mutate** the offsprings to maintain diversity
* **Replace** the current generation with the some fittest individuals and the offsprings
* **end while**

**Implementing GA:**

Implementing GA to solve the problem: What the maximum sum of a bitstring is (a string consisting of only 1s and 0s) of length N?

The trivial answer to the problem is N.

**Initializing the Population**- The problem wanted the maximum sum of bitstrings of length N. So the population will consist of some number of N-length bitstrings. Let N be 1000, so the population is of 1000-length bitstrings. The algo starts with a random list of 1000-length bitstrings. Each Individual of the population is also referred to as a Chromosome. It’s then passed into a set of functions that do some work on the population to produce a new, hopefully, better population. A single step in the pattern is called an *evolution*. Genetic algorithms work via evolutions over multiple generations.

**Evaluating the Population-** A function is made that evaluates the current population based on fitness.This function takes a population, evaluates each individual based on a *fitness function*, and sorts the population based on each chromosome’s *fitness*. In this problem, the fitness of a chromosome is represented by the sum of the bitstring.

**Selecting Parents-** This step is referred to as *selection*. Selection is the process of picking the parents that will be combined to create new solutions. The goal of selection is to pick some parents that can easily be combined to create better solutions.

**Creating Children-** Crossover is analogous to reproduction. It’s a genetic operator that takes two or more parent chromosomes and produces two or more child chromosomes. Thus far, the transformations have produced a list consisting of two 1000-length bitstrings. The task is to produce a population can be passed back into the algorithm function.

**Adding Mutation**- Adding mutation in the offsprings is a vital step to achieving the best solution. Despite initializing the population to a seemingly random distribution, eventually, the parents get too genetically similar to make any improvements during the crossover. This phenomenon is termed as premature convergence. This illustrates the importance of including the mutation step in the algorithm and how vital exploration is. Mutation is similar to the other functions in that it accepts a population as a parameter. Only a small percentage of the population is mutated as well to preserve the progress that’s already been made.

*NOTE:{I am not able to insert this code in ms word for some reason}*

**Implementing GA in a real-world problem by myself:**

I encountered a very interesting video by a YouTube channel named “Vesitarium”. It was about how luck and skill play a role for a person to become an astronaut. Having recently read and learned about Genetic Algorithms, I thought of implementing them on this concept of the video. Let’s assume that 100 people want to become an Astronaut. In order for them to be selected, we calculate their capability level. The more the capable an individual is, the more is their chances of getting selected. Capability turns out is dependent on 2 factors:

• Skill (95%) • Luck (5%)

I wanted to use Genetic Algorithm to simulate the ideal capability level that the candidate must have to be selected into Astronaut training program.

The answer should come out to be 100% Capability (95% Skill + 5% Luck). This was a great way to implement my knowledge and concepts.

**Solution:**

* Importinglibraries**:**
* import numpy as np
* import matplotlib.pyplot as plt
* MUTATAION PROBABILITY=[0.02,0.02]
* Defining a class “RandomVariable”:

The class has the following variables:

    luck: A random real number in [0, 5)

    skill: A random real number in [0, 95)

* class RandomVariable():
* def \_\_init\_\_(self):
* self.luck = 0
* self.skill = 0
* self.\_\_calc\_values()
* def \_\_calc\_values(self):
* self.luck = np.random.random() \* 5
* self.skill = np.random.random() \* 95
* Defining a class “person”:

The class has following variables-

luck: The luck of the person

skill: The skill of the person

capability: luck + skill(out of 100)

* class Person():
* def \_\_init\_\_(self, luck, skill):
* self.luck = luck
* self.skill = skill
* self.capability = luck + skill
* def mate(self, par2):
* global MUTATION\_PROBABILITY
* rv = RandomVariable()
* prob = np.random.rand(5)
* if prob[0] < (1-MUTATION\_PROBABILITY[0])/2:
* new\_luck = self.luck
* elif prob[0] < (1-MUTATION\_PROBABILITY[0]):
* new\_luck = par2.luck
* else:
* new\_luck = rv.luck
* if prob[1] < (1-MUTATION\_PROBABILITY[1])/2:
* new\_skill = self.skill
* elif prob[1] < (1-MUTATION\_PROBABILITY[1]):
* new\_skill = par2.skill
* else:
* new\_skill = rv.skill
* return Person(new\_luck, new\_skill)
* Defining a class “Generation”:

The class has following variables-

**\_\_parents**: Details regarding the sorted previous population

**\_\_number**: The number of generation

**new\_population**: The sorted population obtained by mating of the best among previous population

* class Generation():
* def \_\_init\_\_(self, parents, number):
* self.\_\_parents = parents
* self.\_\_number = number
* self.population = self.\_\_mate()
* def \_\_mate(self):
* population = []
* population.extend(self.\_\_parents[:10])
* for i in range(90):
* par1 = np.random.choice(self.\_\_parents[:30])
* par2 = np.random.choice(self.\_\_parents[:30])
* child = par1.mate(par2)
* population.append(child)
* population = sorted(population,  key=lambda x: -x.capability)
* return population
* def print\_fittest(self):
* print("Generation: {}".format(self.\_\_number))
* print("    Capability: {}".format(self.population[0].capability))
* print("         Skill: {}".format(self.population[0].skill))
* print("          Luck:  {}".format(self.population[0].luck))
* def plot(self):
* plt.figure(figsize=(20, 10))
* plt.scatter(range(1, 101), [i.capability for i in self.population])
* plt.title("Generation " + str(self.\_\_number))
* plt.xlabel("Person ID")
* plt.ylabel("Capability")
* plt.show()
* plt.show()
* **Generating Initial Dataset:** Assuming 100 individuals initially, with completely arbitrary skill and luck level
* population = []
* for i in range(100):
* rv = RandomVariable()
* individual = Person(rv.luck, rv.skill)
* population.append(individual)
* population = sorted(population, key=lambda x: -x.capability)
* print("Generation: 1")
* print("    Capability: {}".format(population[0].capability))
* print("         Skill: {}".format(population[0].skill))
* print("          Luck:  {}".format(population[0].luck))
* plt.figure(figsize=(20, 10))
* plt.scatter(range(1, 101), [i.capability for i in population])
* plt.title("Generation 1")
* plt.xlabel("Person ID")
* plt.ylabel("Capability")
* plt.show()

OUTPUT-

Generation: 1

Capability: 96.79404488832583

Skill: 92.43047373695016

Luck: 4.3635711513756705

Chart, line chart

Description automatically generated

* Further Running a genetic algo to find out the optimum value for selection, which ideally should be 100.
* for i in range(2, 6):
* generation = Generation(population, i)
* generation.print\_fittest()
* generation.plot()
* population = generation.population

***OUTPUTS-***

Generation: 2

Capability: 97.03570170242978

Skill: 93.60709373030451

Luck: 3.4286079721252616

Chart, line chart

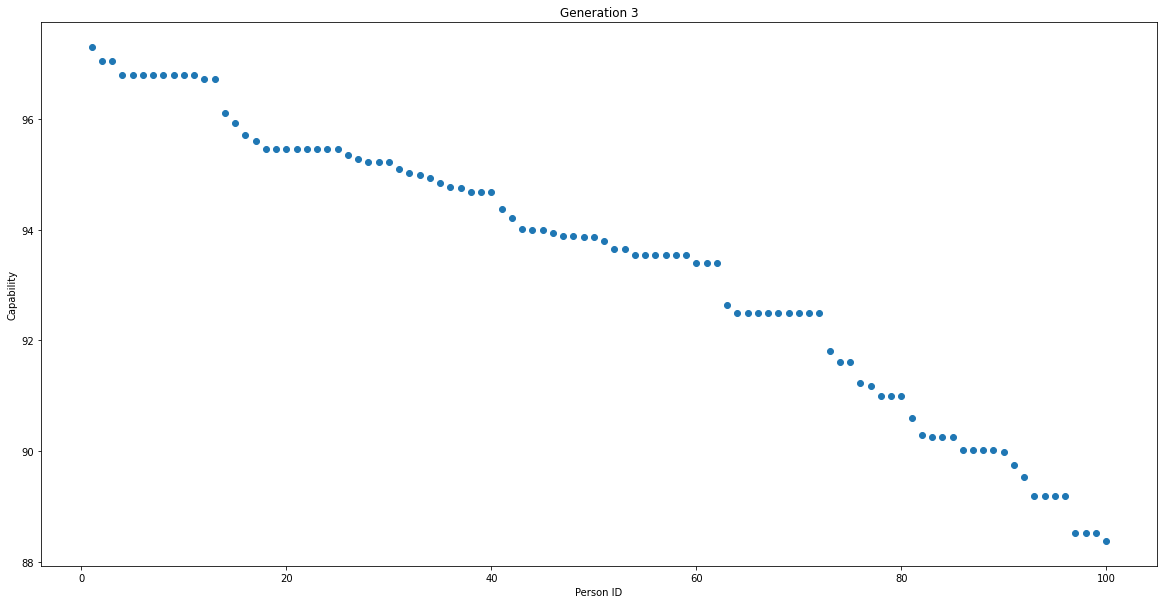
Description automatically generated

Generation: 3

Capability: 97.2940413385364

Skill: 92.34862332924006

Luck: 4.945418009296347

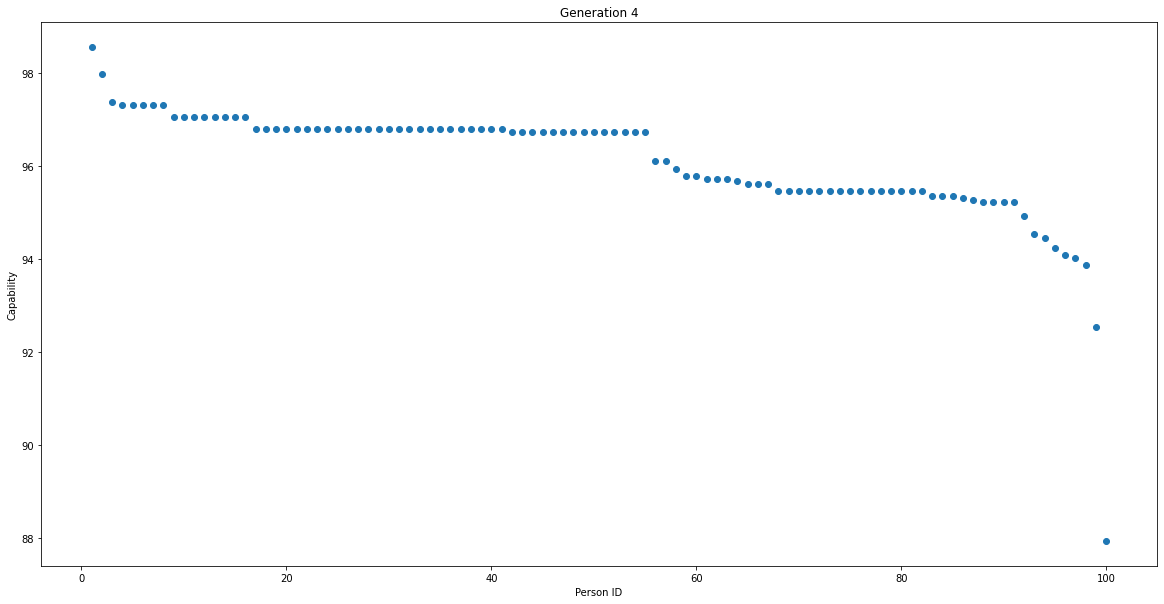


Generation: 4

Capability: 98.55251173960086

Skill: 93.60709373030451

Luck: 4.945418009296347



Generation: 5

Capability: 98.55251173960086

Skill: 93.60709373030451

Luck: 4.945418009296347

Shape, rectangle

Description automatically generated

**OBSERVATION:**

Thus, with each iteration, the capability of the ideal candidate tends to 100. This translates to 5% Luck and 95% Skill. Hence, my code was a success in implementing GA 😊.

**HERE IS THE LINK TO THE GIT REPO OF THE CODE-**

https://github.com/its-harshit/Genetic-Algorithm