Report: Genetic Algorithm (Assignment - 3)

Jaidev Shriram (2018101012), Rohan Grover (2018101017)

April 2020

Contents

1 Introduction

This assignment was about using a genetic algorithm to find an idea set of weights that can form a function which would well fit a hidden data set. This report will explore the logic behind the genetic algorithm along with separate components that were kept in mind to obtain a low error set of features.

2 Summary of Genetic Algorithm

```python
def genetic_algorithm(population):

    i = 0
    j = 0

    while True and i < 10:

        i = i + 1
        j = j + 1

        fittest = population.get_fittest()

        if avg(fittest.fitness) <= 1e5:
            break

        population.sort()

        size = int(len(population) - 1)
        if j > 5 and j < 10:
            size = int(len(population) * (3/4))
        elif j > 10:
            j = 0

        new_population = populate(population, size)
        population = new_population
```

Listing 1: Genetic Algorithm Code

Note: This code has been slightly altered from the submitted code as it is missing DEBUG statements and calls to functions that save the population to a file.

This is a very standard genetic algorithm and was primarily based off of the one taught in class. The code above just depects the terminating condition of the algorithm, along with the initialization of a new population. Our genetic algorithm also does some elitism. This is for our selection algorithm which chooses from the top section of our population. When sorted by fitness, this means that only vectors with high fitness/low error produce children.

The key components of a GA are in our populate function.

The function below runs through these three phases of a genetic algorithm:

1. Selection
2. Crossover
3. Mutation

```python
def populate(population, size):

    new_population = Population()

    for i in range(int(len(population) / 10)):
        temp = population.population[i]
        new_population.population = np.append(new_population.population, temp)

    for i in range(int(len(population) / 10), len(population), 2):

        rand1 = randint(0,  int(size))
        rand2 = randint(0, int(size))

        first_parent = population.population[rand1]
        second_parent = population.population[rand2]

        child = reproduce(first_parent, second_parent)

        child[0] = child[0].mutate()
        child[1] = child[1].mutate()

        new_population.population = np.append(new_population.population, child[0])
        new_population.population = np.append(new_population.population, child[1])

    return new_population
```

Listing 2: Genetic Algorithm's Populate Function

This function chooses top 10% of population as part of elitism. Then the remaining 90% of the population is produced by selecting from the top portion of the population specified by size. We will explain this in more detail in the subsequent sections that explain the algorithm in more detail.

3 A Depiction of the Genetic Algorithm

The diagram at the end of the PDF is a great depiction of the way the algorithm works.

A few things to keep in mind while looking at the diagram:-

1. Due to space constraints, the precision of the values in our vector are kept to two decimal places. Due to this, a lot of mutations that happen in lower decimal places go unnoticed and vectors also tend to look similar after the crossover. However, that is not the case with our actual Genetic Algorithm as we use high precision during it's runtime.
2. Our GA sorts the population by fitness before every iteration. However this has not been done in the diagram as this is a sample of random vectors with no fitness values attached to them
3. It can also be observed that our algorithm never picks the lowest two individuals in our population.This is because our algorithm ignores the bottom quarter of our population (as those are the low fitness vectors after sorting). This is to ensure we get high fitness vectors to cross over with each other more.

4 Fitness Function

One change from the the base generic algorithm was that we assigned a particular error value of the vector as it's fitness. Thus, instead of maximising fitness of our population, our goal was to minimise the fitness as much as possible. However, to avoid confusion in the report, high fitness and low error are equivalent.

With that in mind, we tried multiple fitness functions, and the following was the best one:-

4.1 Average of Train and Validation Errors

$$(train + valid)/2$$

This relatively simple function works well to reduce both errors. To avoid overfitting, a penalty of few lakhs was added to the fitness value when the difference in train and validation errors is more than 3 Lakh. Hence, this function works well at minimizing error while avoiding overfitting.

4.2 Other functions used

We also used

$$train^2 + valid^2$$

at one point but it did not make much of a difference. One function that is great in theory but couldn't be tested as we had already hit a local minima was

$$abs(train - valid)^2 * train$$

5 Selection Logic

First, we sort the population based on fitness. Then we randomly choose two parents from this population, to take part in reproduction - crossover. However, to ensure that the children produced aren't all bad/low fitness, we choose vectors from the first 3/4th of the population. Later, as we noticed that the population was becoming too similar because of a lack of diversity in this portion, we randomly chose from the entire population.

6 Crossover Function

Since this problem uses real numbers, it was important to use an algorithm that uses real encoding.

6.1 Simulated Binary Crossover

```

eta = 10
for i in range(11):
    rand = random()
    beta = 1

    if rand > 0.5:
            beta = (1 / (2. *(1 - rand)))
        else:
            beta = (2. * rand)

        beta **= (1. / (eta + 1.))

        child1.vector = np.append(child1.vector, 0.5 * (((1 + beta) * first_parent.
    vector[i]) + ((1 - beta) * second_parent.vector[i])))
        child2.vector = np.append(child2.vector, 0.5 * (((1 + beta) * second_parent.
    vector[i]) + ((1 - beta) * first_parent.vector[i])))
```

Listing 3: Simulated Binary Crossover

We decided to use a crossover that was not covered in the textbook and referred to this research paper. This algorithm lets us explore the continuous real space much better than algorithms that we considered first (listed in the next subsection). This algorithm lets us explore more of the search space by achieving arbitrary precision in a child that is produced. This crossover's implementation was partially taken from the implementation by the DEAP Framework.

These are some properties of SBX:

1. Average gene values of children are the same as that of parents
2. We can tweak the distance of children from parents - which helps in changing the diversity of a population
3. As per the paper, it is a simulation of single point crossover for real encoded chromosomes.
4. The genetic algorithm does not have to rely on mutations to change values of genes, and crossover helps in exploring search space

6.2 Parameters to Tune

The crossover has a parameter (eta in the code) which is the divergence factor. This has an inverse relation with the 'distance from parents'. Low eta means that children produced may not be very similar to parent. However, high eta means that children produced can be close to parents. This is a very important parameter and was tweaked extensively during various iterations of the program. Intially, to bring diversity, the eta value was set at two. Midway, it was set to 20, to produce more useful children to let the algorithm converge to a value.

6.3 Comparison with Other Crossovers

Some other crossovers chosen include 'Single Point Crossover', 'Double Point Crossover', and 'Uniform Crossover'. These were seen to be bad in general for multiple reasons:

1. The order of weights in the vector tend to be be fixed. The only parameter worth changing is the exact value. For instance, from our initial vector, 5.1e-05 might change to 6.1e-05 but is very unlikely to become 2.3 or 5.1e-08. There are few weights for which order does change but those are very few and have little impact on error.
2. Algorithms meant for binary strings do not help explore search space well as mentioned earlier. This is because they only change ordering, and not the value itself.

Hence, this was found to produce higher fitness vectors in much fewer iterations than 'Single Point Crossover' as stated in literature.

7 Mutations

Mutations are applied with a probability of 0.15 to any **one** feature of every child. This essentially means that every child is mutated with probability 0.15. The main point of mutations is to introduce some randomness and ensure that local minimas are avoided.

The exact mutation function used is:

```
val += (val * np.random.uniform(-0.1, 0.1))
```

Listing 4: Mutation

Since errors change drastically for even changes to the order, we had to ensure that the order was the same, so val is a parameter in it's own mutation (val is multiplied with a number before adding).

The value 0.1 was chosen experimentally as mutations should be enough to change the search space but not change the vector so drastically that it is redundant because of large errors.

Further the probability was chosen arbitrarily and performed well for the most part.

8 Other Hyper Parameters

8.1 Pool Size

A major consideration for pool size was the resources used. Since resources were limited, we wanted to ensure that the pool was not very large as there is no guarantee that the error will drop quicker because of a larger pool. One advantage to a large pool size is diversity in population, so we decided to settle for 20 values. The selection algorithm and mutations help ensure that there is some diversity to the population.

8.2 Selection Pool Size

Initially we were using 1/2 the population (highest fitness ones) to take part in selection. But this performed poorly when we were close to a local minima as the vectors were all nearly identical. Hence, we increased pool size to 3/4th and then alternate between this and choosing from the entire population. We decided to alter every 5 iterations as it seemed to be the best for ensuring diversity in the population while simultaneously producing good children in the long run

9 Statistical Information

9.1 Number of Restarts

No. of restarts: 4

One challenging aspect with this assignment is the nature of the dataset that we are expected to model. Since many local minimas are there, it is hard for a set of mutations to explore a different area of search space of which the slope could lead to global minima. But simple mutations along the way make this hard to do as increasing a weight may result in a higher error but may actually be required for a global optima.

Hence, each time we restarted, we either started with extensively mutated overfit vectors or otherwise (described in the end).

The following are some vectors which converged to local minima and forced us to restart:-
[-2.9693193431543444e-11, 1.8715420139593866, -6.922088493160181, 0.06539782444542824, 0.03870620199603078, 9.28803139883789e-05, -6.01876914689282e-05, -1.2957851278314178e-07, 3.484096383229683e-08, 3.825538145448347e-11, -6.732420176902685e-12] Error : 6.3L

[-4.953649279947332e-14, 0.12119706586017825, -6.445067263467308, 0.08425235331447335, 0.03846886029743811, 8.118911173407719e-05, -6.0187691609169116e-05, -1.313443631190975e-07, 3.484096383229681e-08, 3.997491889644974e-11, -6.732420176902564e-12]

Error: 6.7L

9.2 Iterations for Convergence

On average, on starting, it took around 200 iterations to converge. This means that the rate of drop in error reduced significantly. For instance, after 200 iterations, it takes 100 iterations for average error to drop by around a few thousand.

10 Various Approaches

10.1 Starting Completely Random

Initially, for the first approach, we assumed that we could get to high fitness by randomly initializing within the range -10, 10. This was highly unsuccessful obviously as the weights in the vector are very small.

10.2 Starting with Overfit Vector

We then realized that the overfit vector was a good starting point for an initial population. This vector largely had the ranges / order of the vector is correct - or rather the most sensitive features. This worked great for most part. During 3 "restarts", we were able to start with a population which had a best vector with average error of around 16L (the penalty for overfit was also added). These errors usually dropped to around 10L in very few iterations - always around 25. It then slowly dropped to around 6L. To be precise, the average error was approximately, 6.3L, 6.9L and 7L. Unfortunately, it failed to drop even after reaching a rather low error.

10.3 Using Different Initial Population

We never ran 200 iterations in one go. Instead, we did it incrementally. This means that we ran it for say 20 iterations and then stopped - saving the population to a file the entire time. Hence, if we felt that the population was too similiar, we could make slight tweaks to it.

These tweaks were made by mutating multiple features of the vectors in the population - excluding the top few so that progress is not lost. This has defintely helped ensure that the algorithm can continue exploring the search space well rather than being confined to a small area. Further, we also added mutations of other vectors sometimes to increase the diversity.

For instance, we would add extensive mutations of overfit once again - this is not redundant as by this point, we would've come closer to the local minima.

Diversity is obviously important because the crossover operation and the children produced are directly dependent on the parents. Hence, if the parents are nearly the same, so are the children - which is equivalent to not running it at all.

10.4 Final Approach: Solving Problems with Sign

One fundamental problem with starting with overfit - common to 3 restarts of the problem - is that if the overfit vector had a wrong sign to begin with - all mutated versions would to.

This is a very fundamental problem with the overfit vector. Hence, we wanted to see if there was a possibility that global minima vector would have a different sign. Hence we decided to mutate a few 0 vectors and see what errors were obtained.

This process was done manually - manually change mutations of the 0 vector (all 0s) and see if there is any trend. Surprisingly, some vectors had errors in the 10-20L range. Here, the signs of all weights were not the same as other features. Hence, it was clear that there was an entire search space that is being missed. After manual changes - incrementing and decrementing decimals around a bit - the error significantly dropped. Hence, we knew that this search space had to be explored.

So, we decided to mix our population once again. We took half of our old population (the one that converged at around 6.3L), mutations of this new manually obtained error, and mutations of the overfit vector. After a few iterations, the error started reducing again. This is because of our SBX crossover - which generated children closer to either children. Slowly, the population started tending away from the mutated overfit vectors. Then, the regular algorithm was able to reduce average error from 3L to 2.1L.

This is a clear demonstration of how a good initial population is essential. Normally, a mutation function should be able to not require the former - but with multi variate data, it is not possible to change just one/two weights and expect error to reduce. As stated earlier, we need to make multiple adjustments to a vector to realize an entirely new searchspace.

In the end, our best vector was :

[1.2260147504510547e-13, 1.587170365090799, -0.1648229476835562, 0.04547460805535061, -0.00011647479145275894, -1.508349245000208e-08, 1.6924181792542783e-08, 1.7172775737862141e-09, 9.01145331974921e-12, 1.2196919719308326e-13, 6.011111130590995e-15]

Error: 1.9L

This was a gradual but steady drop from 3+ L. The exact value is unfortunately lost.

11 Additional Tricks

11.1 Manual Tweaks

Initially and at converged points, we manually tweaked the vectors. This was not to obtain a new best vector and was not to minimize test error either. Rather it was to obtain new insight into the data. This was crucial for us to understand how to initialize the population and make a mutation function.

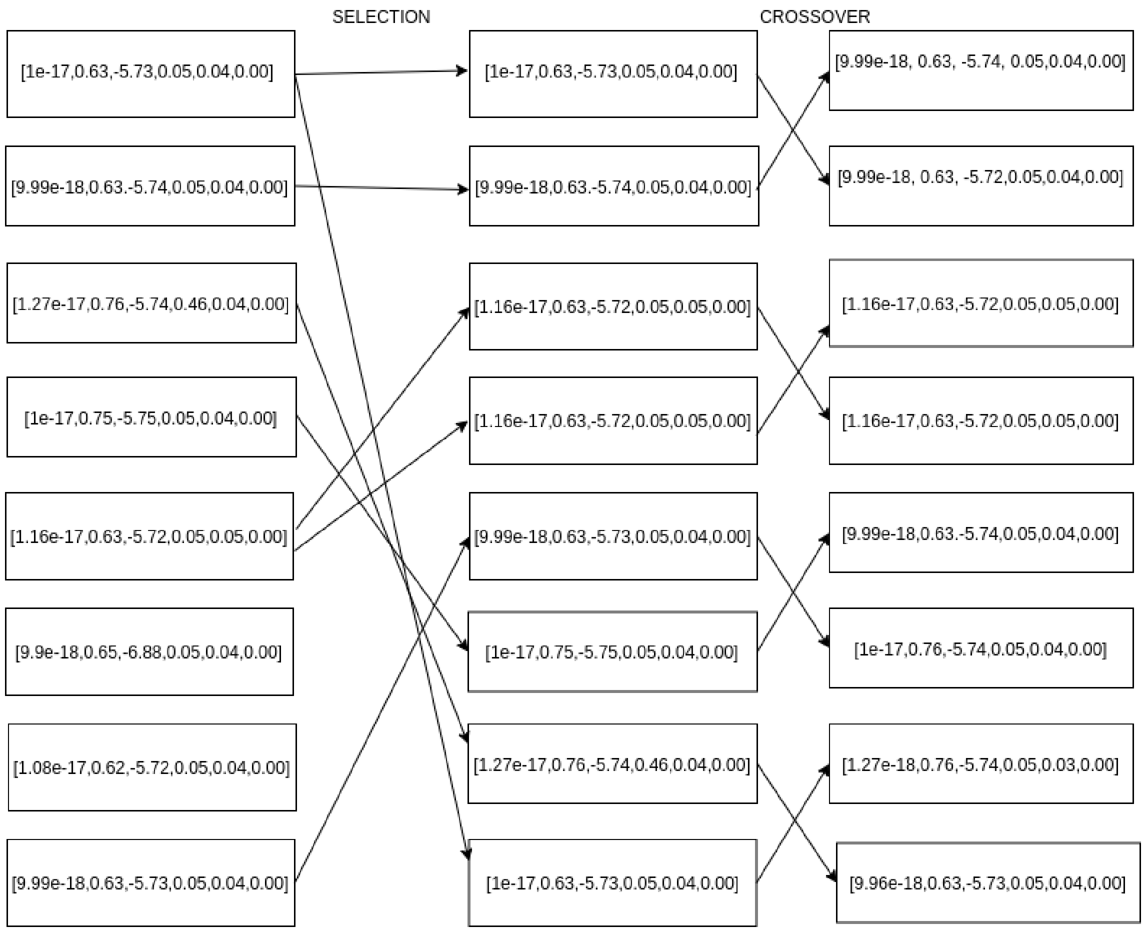
12 How to Run?

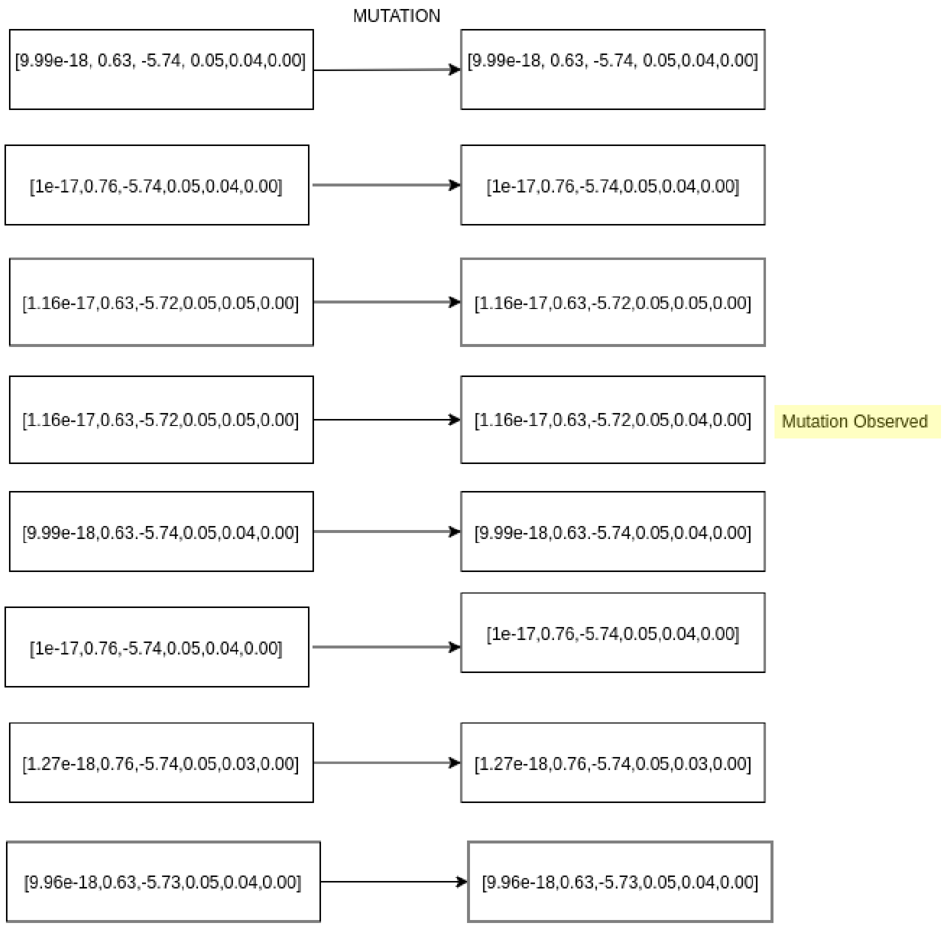
To run the code, just do

```
python3 genetic.py
```

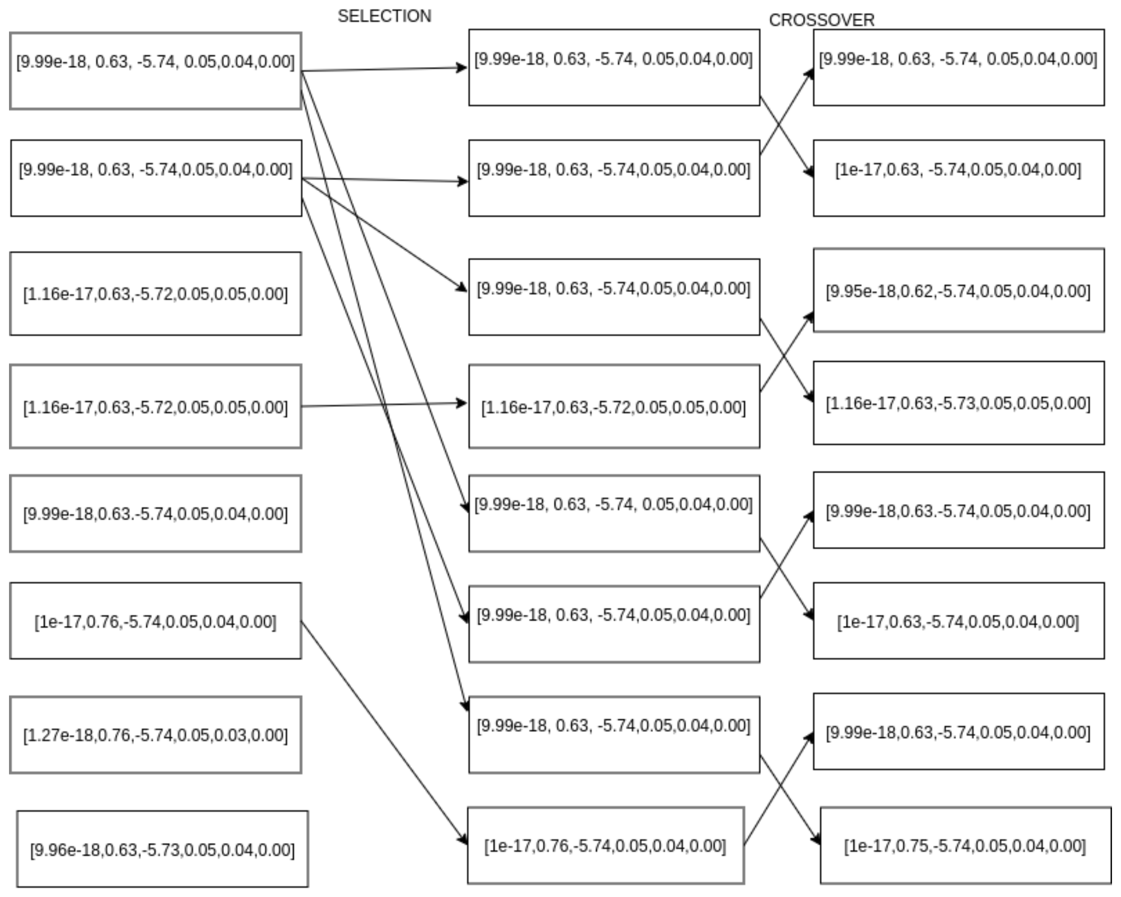
By default, it loads the old population, so last$_trace.datamustalsobeinthesamedirectory.$

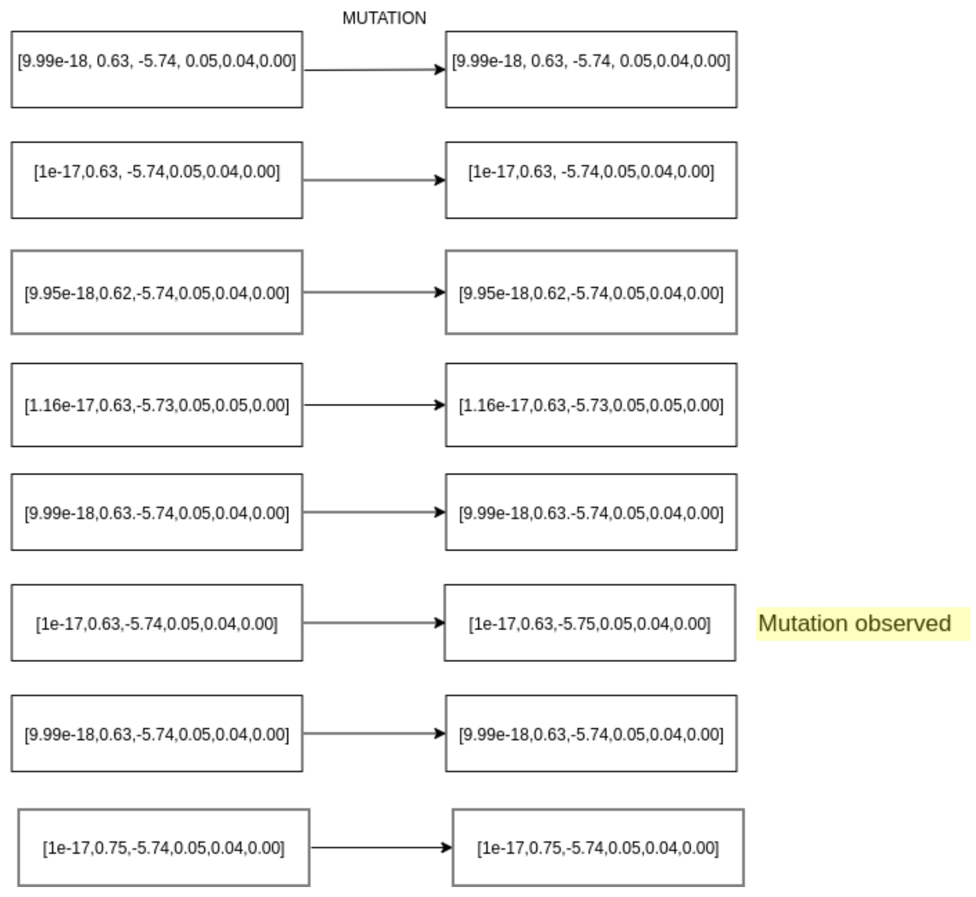
First Iteration


SELECTION
CROSSOVER
[1e-17,0.63,-5.73,0.05,0.04,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1.27e-17,0.76,-5.74,0.46,0.04,0.00]
[1e-17,0.75,-5.75,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[9.9e-18,0.65,-6.88,0.05,0.04,0.00]
[1.08e-17,0.62,-5.72,0.05,0.04,0.00]
[9.99e-18,0.63,-5.73,0.05,0.04,0.00]
[1e-17,0.63,-5.73,0.05,0.04,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[9.99e-18,0.63,-5.73,0.05,0.04,0.00]
[1e-17,0.75,-5.75,0.05,0.04,0.00]
[1.27e-17,0.76,-5.74,0.46,0.04,0.00]
[1e-17,0.63,-5.73,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.99e-18, 0.63, -5.72,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1.27e-18,0.76,-5.74,0.05,0.03,0.00]
[9.96e-18,0.63,-5.73,0.05,0.04,0.00]

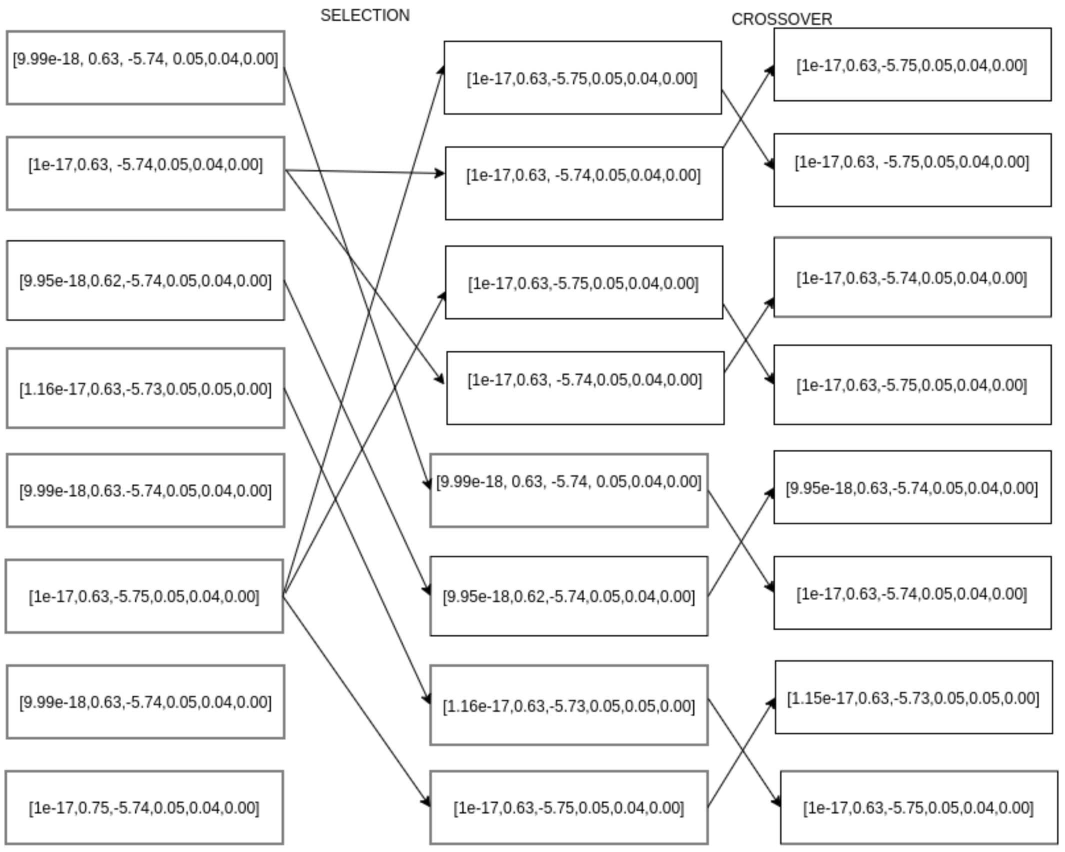

MUTATION
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.04,0.00]
Mutation Observed
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1.27e-18,0.76,-5.74,0.05,0.03,0.00]
[1.27e-18,0.76,-5.74,0.05,0.03,0.00]
[9.96e-18,0.63,-5.73,0.05,0.04,0.00]
[9.96e-18,0.63,-5.73,0.05,0.04,0.00]

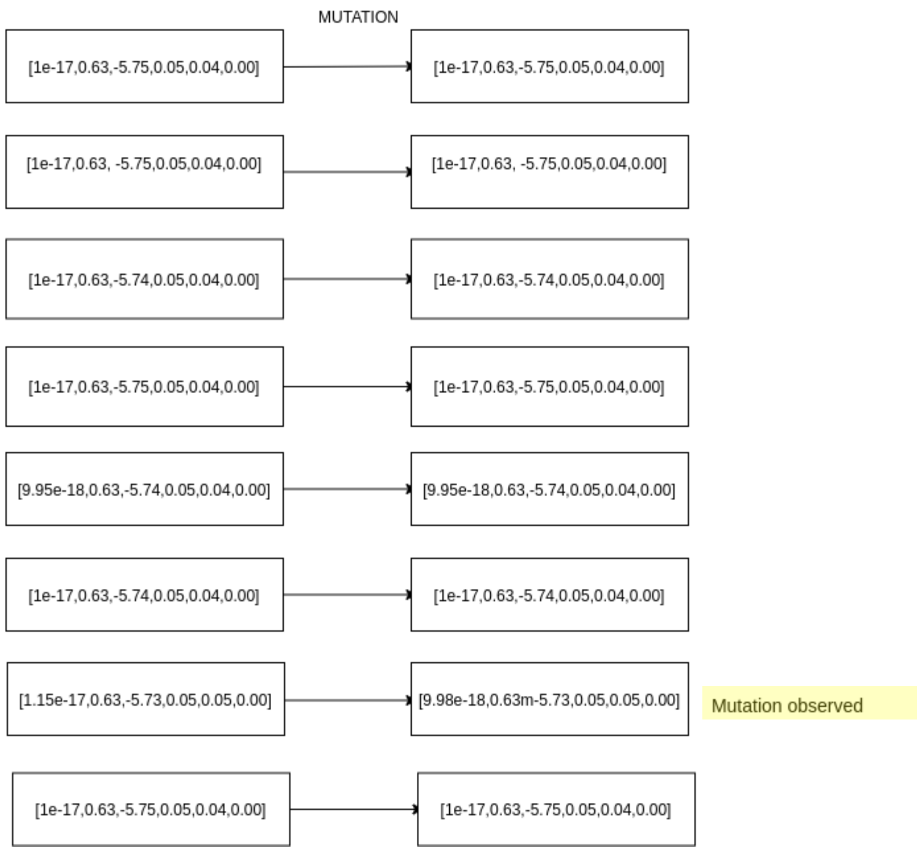
Second Iteration


SELECTION
CROSSOVER
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[1.27e-18,0.76,-5.74,0.05,0.03,0.00]
[9.96e-18,0.63,-5.73,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.72,0.05,0.05,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74,0.05,0.04,0.00]
[1e-17,0.76,-5.74,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[9.95e-18,0.62,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.73,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[9.99e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.75,-5.74,0.05,0.04,0.00]


MUTATION
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[9.95e-18,0.62,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.73,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[9.99e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.75,-5.74,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[9.95e-18,0.62,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.73,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
Mutation observed
[9.99e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.75,-5.74,0.05,0.04,0.00]

Third Iteration


SELECTION
CROSSOVER
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[9.95e-18,0.62,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.73,0.05,0.05,0.00]
[9.99e-18,0.63.-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[9.99e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.75,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63, -5.74,0.05,0.04,0.00]
[9.99e-18, 0.63, -5.74, 0.05,0.04,0.00]
[9.95e-18,0.62,-5.74,0.05,0.04,0.00]
[1.16e-17,0.63,-5.73,0.05,0.05,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63, -5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[9.95e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1.15e-17,0.63,-5.73,0.05,0.05,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]


MUTATION
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63, -5.75,0.05,0.04,0.00]
[1e-17,0.63, -5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[9.95e-18,0.63,-5.74,0.05,0.04,0.00]
[9.95e-18,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1e-17,0.63,-5.74,0.05,0.04,0.00]
[1.15e-17,0.63,-5.73,0.05,0.05,0.00]
[9.98e-18,0.63m-5.73,0.05,0.05,0.00]
Mutation observed
[1e-17,0.63,-5.75,0.05,0.04,0.00]
[1e-17,0.63,-5.75,0.05,0.04,0.00]