Machine, Data and Learning

Assignment 3

Team Number - 66
Satyam Viksit Pansari (2018101088)
Anishka Sachdeva (2018101112)

Problem Statement

You have been given coefficients of features (a vector of size 11) corresponding to an overfit model and your task is to apply the genetic algorithm in order to reduce the overfitting i.e. generalize the model so that the model performs better on unseen data. To help you with the formulation of your fitness function, you will be allowed to query and check how your coefficients perform on the training and the validation set using the server provided.

Given Overfit Vector -

[0.0, 0.1240317450077846,-6.211941063144333, 0.04933903144709126, 0.03810848157715883, 8.132366097133624e-05, -6.018769160916912e-05, -1.251585565299179e-07, 3.484096383229681e-08, 4.1614924993407104e-11, -6.732420176902565e-12]

Initial Errors -

Train - 79569.63536912124
Validation - 3625792.834235452

Genetic Algorithm Explanation in General

(Taken from GeeksForGeeks)

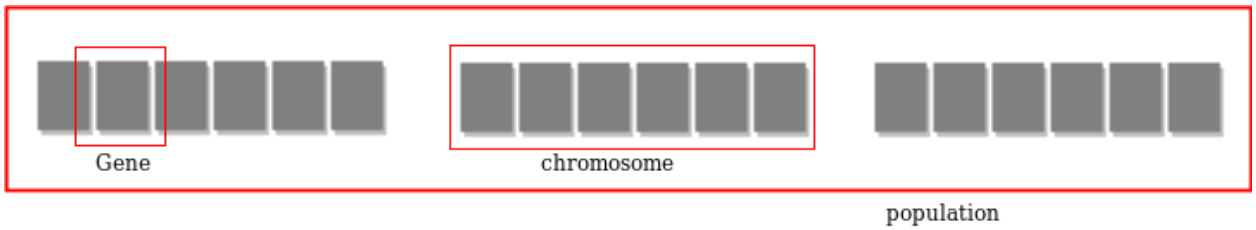
Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to the next generation. In simple words, they simulate “survival of the fittest” among the individuals of the consecutive generation for solving a problem. Each generation consists of a population of individuals and each individual represents

a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.
Genetic algorithms are based on an analogy with genetic structure and behavior of the chromosome of the population. Following is the foundation of GAs based on this analogy –

- Each Individual in the population competes for resources and mate.
- Those individuals who are successful (fittest) then mate to create more offspring than others.
- Genes from the “fittest” parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.
- Thus each successive generation is more suited for their environment.

1. Search Space

The population of individuals are maintained within search space. Each individual represents a solution in the search space for the given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



2. Fitness Score

A Fitness Score is given to each individual which shows the ability of an individual to “compete”. The individual having optimal fitness scores (or near optimal) are sought.

The GAs maintain the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chances to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating a new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

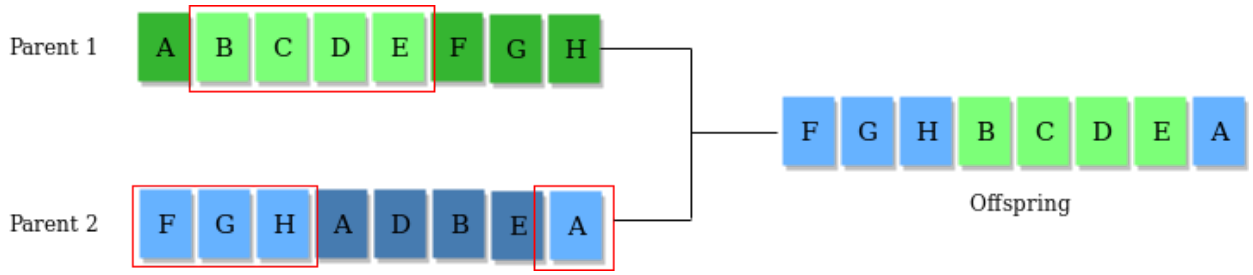
Each new generation has on average more “better genes” than the individual (solution) of previous generations. Thus each new generation has better “partial solutions” than previous generations. Once the offspring produced have no significant difference than offspring produced by previous populations, the population converges. The algorithm is said to be converged to a set of solutions for the problem.

3. Operators of Genetic Algorithms

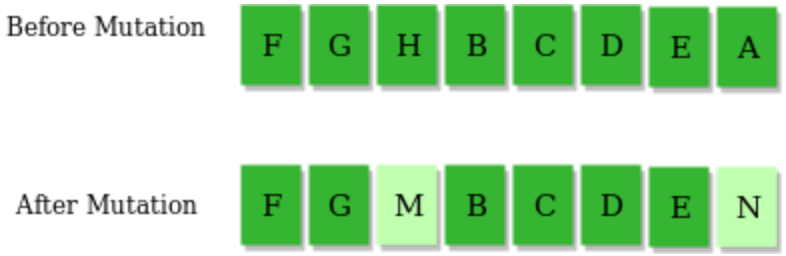
Once the initial generation is created, the algorithm evolve the generation using following operators –

1. **Selection Operator**: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to the successive generations.

2. **Crossover Operator**: This represents mating between individuals. Two individuals are selected using the selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).



3. **Mutation Operator**: The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence.



Libraries Used

1. Os

The OS module in python provides functions for interacting with the operating system. OS, comes under Python's standard utility modules. This module provides a portable way of using operating system dependent functionality.
In the code , we have used it to create the "models.txt" file to analyse the models and their corresponding error values.

2. Random

Functions in the random module depend on a pseudo-random number generator function random().
Used random.choice, random.random, random.uniform.

3. Requests

The requests module allows you to send HTTP requests using Python. The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

4. Json

Python has a built-in package called json, which can be used to work with JSON data. JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation.

Genetic Algorithm we used

Some Key Points:

1. Here the fitness function is basically the cost function for us. In the original algorithm, they try to maximize the fitness score. But in our genetic algorithm, we are trying to minimise the fitness score and hence it refers to minimising the cost function.
2. We are trying to minimise the errors. So here the cost function is basically minimising the two errors.
3. **We had adopted a main strategy : Whenever our model had reached a local minima and wasn't converging, we increased randomness in our genetic algorithm so that we could get a vector out of the local minima. Whenever our model came out of local minima, we decreased the randomness and worked upon our probability aspect.**

    For example : Selecting parents randomly when our model hit the local minima. And introduced the probability array where the best two individuals out of the top 50% individuals of the previous population are more likely to get selected when our model was not at any local minima.

Our genetic algorithm was mostly the same as discussed above.
Some of the minor changes/addition we made to the original genetic algorithm are:
We took the initial population in four different ways:

1. Took the starting vectors to be entirely random i.e. generated random genes for all the chromosomes.
2. Took the first individual to be the overfit vector given to us and generated other vectors randomly.
3. Took the first individual to be the overfit vector given to us and generated other vectors with respect to the genes of the overfit vector.
4. Took the initial population to be the top best vectors of the iterations we ran (as we used a python script to redirect the vectors generated along with their fitness function to keep a track of the vectors generated so far).

We introduced the concept of probabilities at various points in our algorithm such as:

1. Crossover
2. Mutation
3. Selecting parents for child chromosomes from the previous generation

Implementation in the code

Code snippets have been added for the ease of readability and understandability

We merged moodle_client.py file with our genetic algorithm.

We created a class called **Individual** which had the following:

1. Parameterised Constructor:

- Receives an array which is an individual in the population named as chromosome.

- Using the above array, we called the **get_errors** function(written outside the class Individual) to get the values of training and validation error from the server.
- Written the vectors along with their errors in the files created in the code to do the analysis of the data points/vectors.

```
class Individual(object):
    def __init__(self,chromosome):
        global ID
        global file
        self.chromosome = chromosome
        self.err_value = get_errors(ID,self.chromosome)
        file.write(str(self.chromosome ))
        file.write(" ")
        file.write(str(self.err_value))
        file.write("\n")
        file2.write(str(self.chromosome ))
        file2.write(" ")
        file2.write(str(self.err_value))
        file2.write("\n")
        self.fitness = self.cal_fitness()
```

2. Functions:

- **Mate Function**
  - Used for crossover and mutation.
  - Contained the probability aspect for crossover and mutation.
- **Fitness Function**
  - Contained the method of calculating fitness score for each vector.

```
def cal_fitness(self):
    fitness = (self.err_value[0] + self.err_value[1])
    return fitness
```

- **Create_Gnome Function**
  - Not contained in the class Individual. This function is used to create the initial population by appending the genes to form a vector of size 11.

In **Main()**, we have the following:

We took different population sizes ranging from 20 to 35.

1. We created an array of objects of class Individual called **Population** which stored the individual and its corresponding fitness score.
2. The initial population gets calculated in the main function using the create_gnome function.

3. We now begin with the iterations.
4. Firstly, the population array is sorted in descending order according to the fitness score.
5. Total iterations is the number of generations created until the model converges.
6. Now, we select the parents for creating the next generation individual. We take random two individuals from the top 50% of the individuals from the previous generation to create the child.
7. We take top 10% individuals straight into the next generation.
8. We submitted 10% of our best individuals (which had the least fitness score) on the server using the **submit** function.

The whole algorithm can be summarized as:

1) Initialize populations.
2) Determine fitness of population.
3) Until convergence, repeat:
    a) Select parents from the previous population.
    b) Crossover and generate new population.
    c) Perform mutation on new population.
    d) Get the training and validation errors for the child created.
    e) Calculate fitness for the new population.

Percentage of vectors we submitted for each iteration

We submitted the top 10% to 20% of the vectors depending upon the population size which had the least fitness scores in each iteration.

Later on we realised that as time proceeded, the same vectors were getting submitted because the number of good vectors generated started to decrease. And hence later on we didn't submit until we got good vectors and our model started to converge.

Approaches used for tuning the Feature Coefficients

(Mentioned the major parameters changed)

1. Approach 1:

- ### Initial Population:
  We took the initial population consisting of random individuals. The first individual was taken as the overfit model given to us. And the rest were generated using the random function of Python. All the 11 genes of a gnome (individual) were "**random values between -10 and 10".**
- ### Child Population(Next Generation):
  The child population was a mix of previous population and new individuals as well. The fitness of the individuals were sorted in ascending order to get the individuals with low fitness value (low error value) to be used for the child population.

```
population = sorted(population, key = lambda x:x.fitness)
```

For the new generation (child population), the **top 20%** of the individuals were taken from the previous population and the remaining population was formed by mutation. The 2 parents were chosen randomly from the **top 50%** of the previous population and then were further mutated.

```
for i in range((POPULATION_SIZE-int((80*POPULATION_SIZE)/100))):
    child_population.append(population[i])
```

```
parent1 = random.choice(population[:int((50*POPULATION_SIZE)/100)])
parent2 = random.choice(population[:int((50*POPULATION_SIZE)/100)])
```

- **Crossover and Mutation Function:**

  Crossover and mutation was done on the basis of the probability assigned. We generated a random probability using random.random() function. And depending on the probability value, we chose either mutation or crossover.
  - With probability < 0.45, we took a gene from parent 1.
  - With probability < 0.90, we took the gene from parent 2 (i.e., with equal probabilities, we took a gene from either parent).
  - Else we applied mutation by generating a random gene using random.choice().
- **Fitness Score:**

  Minimising the sum of two errors (Training error and Validation error) i.e.
  Fitness Score = Training Error + Validation Error

```
def cal_fitness(self):
    fitness = (self.err_value[0] + self.err_value[1])
    return fitness
```

- **Observations:**
  - Even after running for around 20-30 generations no satisfactory model was achieved.
- **Inference:**
  - A complete random start is pointless and will take a lot of time to converge. It's better to use the given overfit model top generate initial population rather than starting with complete random individuals.

2. Approach 2:

- **Initial Population:**

  The first individual was taken as the overfit model given to us. Then the initial population was taken with respect to the first individual. We generated the individuals by generating a random number between -1 and 1 and multiplied it with **10^(-10)**. Then we added this small noise in each gene of the first individual and created new genes, thus forming a new individual.

```
gene = random.uniform(0.0,1.0)
for i in range (len(gnome)):
    gnome[i]=gnome[i]+(gene*(1e-10))
```

- **Child Population(Next Generation):**
  Same as above.
- **Crossover and Mutation Function:**
  We changed the crossover and mutation by introducing the concept of noise.
  - With probability < 0.45, we either took the exact gene from parent 1 or a mutated version of it. So with probability < 0.0125, we took the gene from parent 1 and added a noise of the order (1e^(-13) * gene of parent 1) . With probability < 0.45, we took the gene from parent 1 without adding noise.

  - With probability < 0.90, we either took the exact gene from parent 2 or a mutated version of it. So with probability < 0.4625, we took the gene from parent 2 and added a noise of the order (1e^(-13) * gene of parent 2) . With probability < 0.90, we took the gene from parent 2.

```
if prob < 0.45:
    if prob < 0.0125:
        gene = random.uniform(-1.0,1.0)
        gp1=gp1+(gene*(1e-13))
    child_chromosome.append(gp1)
elif prob < 0.90:
    if prob < 0.4625:
        gene = random.uniform(-1.0,1.0)
        gp2=gp2+(gene*(1e-13))
    child_chromosome.append(gp2)
```

  - Else we applied mutation by generating a random gene using random.choice().
  - So probability distribution is :
    - Probability of gene to be un mutated from parent 1 = 0.4475
    - Probability of gene to be un mutated from parent 2 = 0.4475
    - Probability of gene to be mutated from parent 1 = 0.0125
    - Probability of gene to be mutated from parent 2 = 0.0125
    - Probability of gene to be a random number = 0.1
- **Fitness Score:**
  Same as above.
- **Observation:**
  - After running the code for a few times with around 20 generations each time with population size 35, we reached models with the sum of errors around 25-30 lakhs.

- **Inference:**
    - The noise used for the initial population didn't work out. (Earlier we had also tried large values of noise of the order 10^(-4) and so. They didn't work so we tried with 10^(-10)) So we tried 10^(-13) because it was all about the sensitivity of the coefficients and hence wanted to try even smaller noise to tune them.

3. Approach 3:

- **Initial Population:**
  The first individual was taken as the overfit model given to us. Then the initial population was taken with respect to the first individual. We generated the individuals by generating a random number between -1 and 1 and multiplied it with **10^(-13)**. Then we added this small noise in each gene of the first individual and created new genes, thus forming a new individual.
- **Child Population:**
  Same as above.
- **Crossover and Mutation Function:**
  Changed the probabilities of mutation and crossover.

```
prob = random.random()
if prob < 0.45:
if prob < 0.4:
    if prob < 0.0125:
        gene = random.uniform(-1.0,1.0)
        gp1=gp1+(gene*(1e-13))
    child_chromosome.append(gp1)
elif prob < 0.90:
    if prob < 0.4625:
elif prob < 0.80:
    if prob < 0.4125:
        gene = random.uniform(-1.0,1.0)
        gp2=gp2+(gene*(1e-13))
    child_chromosome.append(gp2)
```

- **Fitness Score:**
  Same as above.
- **Observation:**
    - Not much improvement in the data.
- **Inference:**
    - Major change was required as we suspected to be stuck in a local minima.Thus we decided to work on our fitness function.

- We decided to take the difference of both the errors into consideration because there were many vectors with sum of errors equal but had a lot of different difference values. Our motive was to pick up the vectors which have less difference between the validation and training error.

4. Approach 4:

- **Initial Population:**
  We took the initial population as the best individuals of the previous generations.
- **Child Population:**
  Same as above.
- **Crossover and Mutation Function:**
  Same as above.
- **Fitness Score:**
  Minimising the sum of errors and modulus of the difference of the errors i.e.
  Fitness Score = (Training Error + Validation Error) + | Training Error - Validation Error |
  This was done so that the fitness score takes into consideration the difference of the errors and tries to select the individuals with low magnitude of difference of errors over the ones which have a high magnitude of difference of errors.

```
(self.err_value[0] + self.err_value[1]) + abs(self.err_value[0] - self.err_value[1])
```

- **Observation:**
  - After running the code 3-5 times for around 20 generations with population size around 30 with few tuning parameters(like probability distribution).We reduced sum of errors to around 22 lakhs and both the errors were of similar order(i.e. Not much difference.
- **Inference:**
  The above fitness function worked out very well for us as expected. We tried to run the code using this fitness function with some combinations of minor changes. Then when the difference got controlled, we decided to make the fitness score more sensitive towards the summation of errors than the absolute difference of errors.

5. Approach 5:

- **Initial Population:**
  Same as above.
- **Child Population:**
  Same as above.
- **Crossover and Mutation Function:**
  Same as above.
- **Fitness Score:**
  Minimising the constant times sum of errors and modulus of the difference of the errors i.e.
  Fitness Score = A(Training Error + Validation Error) + | Training Error - Validation Error |
  where A is some constant value.
  This was done to make the fitness score more sensitive towards the sum of errors.

```
def cal_fitness(self):
    fitness = 2*(self.err_value[0] + self.err_value[1]) + abs(self.err_value[0] - self.err_value[1])
    return fitness
```

- **Observation:**
  - Sum of errors further reduced top around 20 lakhs but after that no change in fitness function further helped.
- **Inference:**

  We realised that the errors that we received had the similar difference values .. And training and validation errors started to approach equal values. And hence we decided to remove the difference factor from the fitness score calculation after running the code for a few times and work on further minimizing the sum of errors .

6. Approach 6:

- **Initial Population:**

  Same as above.
- **Child Population:**

  Same as above.
- **Crossover and mutation function:**

  Same as above.
- **Fitness score (Same as Approach 1):**

  Changed back to the sum of errors i.e. trying to minimize the sum of errors.
  Minimising the sum of two errors (Training error and Validation error) i.e.
  Fitness Score = Training Error + Validation Error
- **Observation:**
  - It was of no use. We got different values but sum still remained around 20 lakhs.
- **Inference:**
  - We had reached stalemate situation so an analysis for data gathered till now was required.

7. Approach 7:

Analysis of data points.

- **Observation:**

  We observed that the errors were more sensitive for higher indices and less for lower ones.
- **Inference:**

  We decided to mutate the genes in such a way that the impact of mutation is more on the starting genes (from indices 0-3) and less on the remaining genes.

8. Approach 8:

- **Initial Population:**
  Same as above.
- **Child Population:**
  Same as above.
- **Crossover and mutation function:**
  We took the index factor into consideration. So we generated a random number say R1 between (index + 1, index + 7) and then created a new gene by dividing a random number from -1 to 1 by 10 ^ (R1) and adding it to the parent gene.
  So our basic motive was, as the index increases, the denominator value increases so that the overall value of the fraction decreases and hence creates less impact of mutation on the gene.

```
def mate(self, par2):
        child_chromosome = []
        i=0
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            prob = random.random()
            if prob < 0.5:
                if prob < 0.1:
                if prob < 0.125:
                    if prob < 0.066:
                        random_float=random.uniform(10,20)
                        gene=gp1+(random.uniform(-1,1))/(10**random_float)
                    else:
                        random_float=random.uniform(5,10)
                        gene=gp1+(random.uniform(-1,1))/(10**random_float)
                    if i <3 :
                        if prob<0.5:
                            gene=random.uniform(-10.0,10.0)
                        else:
                            gene=gp1+random.random(-1,1)
                    random_float=random.uniform(i+1,i+6)
                    gene=gp1+(random.uniform(-1,1))/(10**random_float)
                    gp1=gene
                child_chromosome.append(gp1)
            else:
                if prob < 0.6:
                if prob < 0.625:
                    if prob < 0.666:
                        random_float=random.uniform(10,20)
                        gene=gp2+(random.uniform(-1,1))/(10**random_float)
                    else:
                        random_float=random.uniform(5,10)
                        gene=gp2+(random.uniform(-1,1))/(10**random_float)
                    if i <3 :
                        if prob<5.5:
                            gene=random.uniform(-10.0,10.0)
                        else:
                            gene=gp2+random.random(-1,1)
                    random_float=random.uniform(i+1,i+6)
                    gene=gp2+(random.uniform(-1,1))/(10**random_float)
                    gp2=gene
                child_chromosome.append(gp2)
            i=i+1
        return child_chromosome
```

- **Fitness score:**

  Same as above.

- **Observation:**

  Here, our summation of training and validation error decreased by 20K only and reached to . But we noticed that we were getting values of the order 10 ^ (-x) where x was varying

very much. This was impacting the denominator heavily and the approach didn't work out much because the denominator was getting high or low irrespective of the index and was becoming much dependent on the random value selected for the numerator.

- **Inference:**
  We decided to change the numerator parameter which has been discussed in the next approach.

9. Approach 9:

- **Initial Population:**
  Same as above.
- **Child Population:**
  Same as above.
- **Crossover and mutation function:**
  This random numerator value made sure that the probability of numerator value being of the order 10^ (-x) is very less as the probability of selecting 0 from randint(-9,9) is 1/19 and any other value is 18/19.

```
random_float=random.uniform(i+1,i+7)
gene=gp1+(randint(-9,9) + random.uniform(-0.999999999999999,0.999999999999999))/(10**random_float)
```

- **Fitness score:**
  Same as above.
- **Observation:**
  The summation of training and validation errors reached around **14.5 lakhs.**
- **Inference:**
  We wanted to try the sensitivity of the index values on the coefficients. So we decided to slightly vary the denominator value range.

10. Approach 10:

- **Initial Population:**
  Same as above.
- **Child Population:**
  Same as above.
- **Crossover and mutation function:**
  The only change in this approach and the previous approach is the range of the random_float value. Range now taken was (index+1,index+9).

```
random_float=random.uniform(i+1,i+9)
gene=gp1+(randint(-9,9) + random.uniform(-0.999999999999999,0.999999999999999))/(10**random_float)
```

- **Fitness score:**
  Same as above.
- **Observation:**

The summation of training and validation errors reached around **13.9 lakhs.**
We noticed that this approach worked great and realised how gene values for our vectors were so related to the index.
Also, till now we were picking parents randomly from the top 50% of the previous population. We observed that the vectors that were getting selected for being the parents were most of the useless which had more fitness values.

- **Inference:**
  We decided to introduce the concept of **probability** for picking up the parents.

11. Approach 11:

- **Initial Population:**
  Initial Population size decreased to 10.
- **Child Population:**
  Introduced the concept of probability array. Using that, the 2 parents with less fitness score among the top 50% of the individuals were selected for becoming the parents.
  We calculated the weight corresponding to an Individual as:
  Weight(Individual) = Fitness score of all the individuals / Fitness of that particular Individual.
  So less the fitness of an individual, more the weight and hence more chance of that individual to become the parent.

```
probablity_array=[]
fitness_sum=0
for i in range(int((50*POPULATION_SIZE)/100)):
    fitness_sum=fitness_sum+population[i].fitness
for i in range(int((50*POPULATION_SIZE)/100)):
    probablity_array.append(fitness_sum/population[i].fitness)
for _ in range(int((80*POPULATION_SIZE)/100)):
    parents = random.choices(population[:int((50*POPULATION_SIZE)/100)],weights=probablity_array,cum_weights=None,k=2)
```

- **Crossover and mutation function:**
  Same as above.
- **Fitness score:**
  Same as above.
- **Observation:**
  - We converged . It improved till 13 lakhs and then it converged.
- **Inference:**
  We decided to further change our mutation function.

12. Approach 12:

- **Initial Population:**
  Same as above
- **Child Population:**
  Same as above.
- **Crossover and mutation function:**
  Introduced the concept of **scaling.**

The gene which got selected for mutation got multiplied by a number between (1- x/100) and (1 + x/100) Where we varied x from 1 to 10..

- **Fitness score:**
  Same as above.
- **Observation:**
  On trying this approach for an entire day with different values of x, our model converged at a sum of errors around 11 lakhs. We also tried to add some noise(of order 10 e-14) considering some probability but that also didn't work.
- **Inference:**
  So we concluded that our model had hit a local minima.

13. Approach 13(to escape minima):

- **Initial Population:**
  We picked up one best data we had and one was randomly chosen from a set of suitable data (whose sum of errors was around 15-20 lakhs).
  Now rest of the population was generated using the function ((x*g1)+(y*g2))/(x+Y) where x and y were random.and g1 and g2 were genes of the 2 data sets.
- **Child Population:**
  Same as above.
- **Crossover and mutation function:**
  - **Part-A:**
    - Temporarily(just for this approach) removed the concept of weights to choose parents.
    - Child gene was formed by g= ((x*g1)+(y*g2))/(x+Y) function.
  - **Part-B:**
    - Tried scaling by 100 percent.
- **Fitness score:**
  Same as above.
- **Observation:**
  On a long wait and various tuning of parameters we got a vector which gave us the summation of errors around 6 lakhs.
- **Inference:**
  - Now that we had this data point we no longer needed to work toward escaping the minima we could go back to improving this model.

14 . Final Approach

- We picked this model and went back to approach 12 and tried various things :
  - We started creating the initial population through scaling the genes of our best model.
  - Introduced the concept of combination of noise addition (of order 10e-14) and scaling for mutating the genes.
  - We again converged around 5 lakhs so once again we went back to approach 13 and then again to 12.
  - Finally we have been able to reached here-
  - [200036.04781253356, 145751.085462745] 345787.13327527855

THE VECTOR WE THINK SHALL PERFORM BEST ON THE TEST DATA

Error values - [200036.04781253356, 145751.085462745]

Sum - 345787.13327527855

- We think that the vector where the summation of training and validation error was the least with not a very big difference and training error > validation error would give the least test error and hence will perform best on the test data.
- Reason for this assertion(training error>Validation error) lies in our assumption that training data set is much bigger than validation set an in our algorithm we have not used validation and training set and differently and hence we don't go by the traditional expectation for training error<Validation error.

Major heuristics that didn't work for us

- Generating fully random genes for child chromosomes without taking noise into consideration.
- Noise of 10^(-10) to the gene of an individual didn't work out much.
- Picking random parents from the previous population for generating child chromosomes.
- Scaling factor of 10% during mutation and crossover.
- After some time, scaling of the genes by some x% didn't have any impact.

Diagrams showcasing our first three iterations

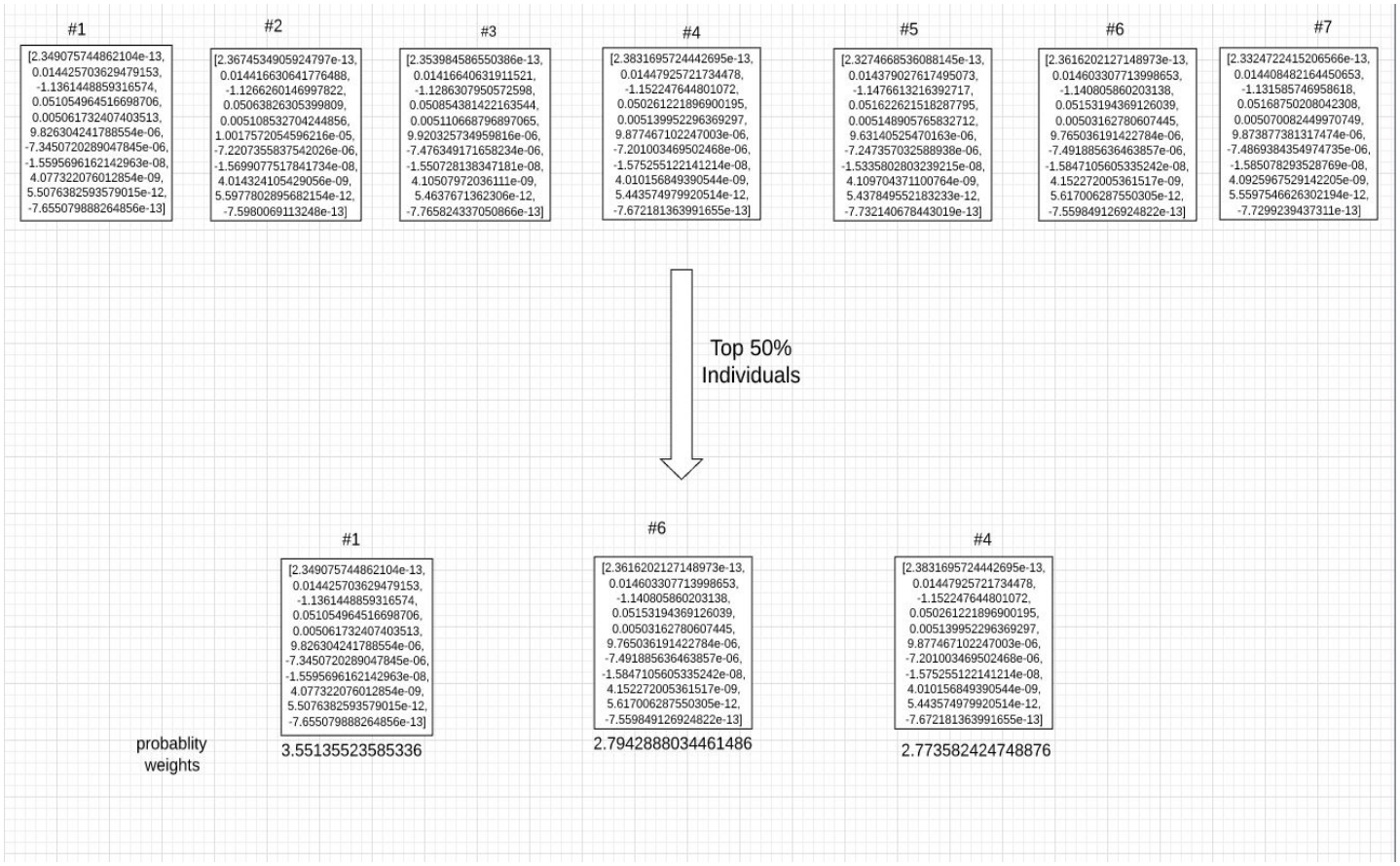
How to study our diagrams:

1. Depiction of initial population for each generation:
    - Individuals of generation 1 are marked with a **single hash.**
    - Individuals of generation 1 are marked with a **double hash.**
    - Individuals of generation 1 are marked with a **triple hash.**
2. Depiction
3. Depiction of each child:
    - Two parents selected for mutation.
    - Arrows from **parents** to the **child without mutation** depicts that the genes of those indices were taken from that particular parent.
    - Arrow from **child without mutation** to **child with mutation** depicts that the genes of those particular indices got mutated.
    - 2 child chromosomes are directly inherited from the previous population depending on the weights calculated using fitness scores.
4. Mutation Function for the Diagram: Scaling by 1% of the genes.

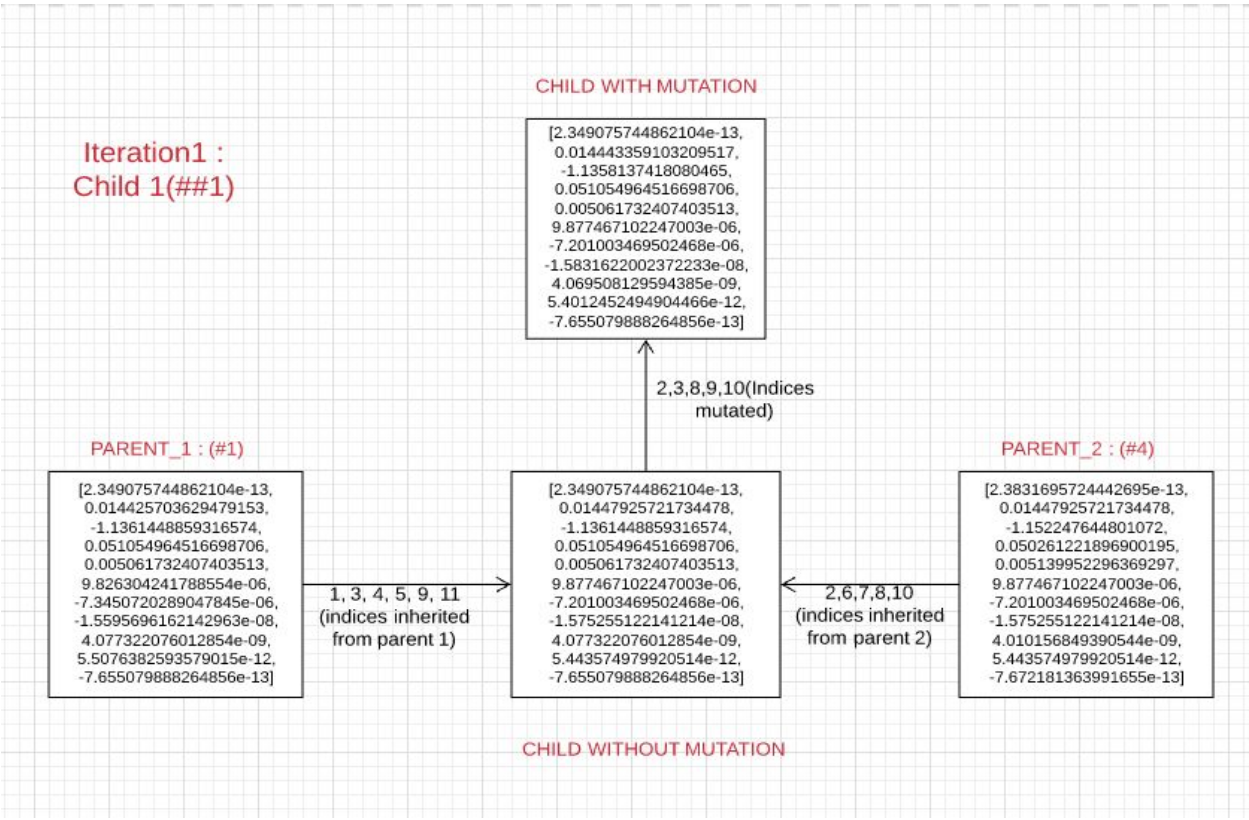
1. Our Population size : 7

2. Initial Individuals :

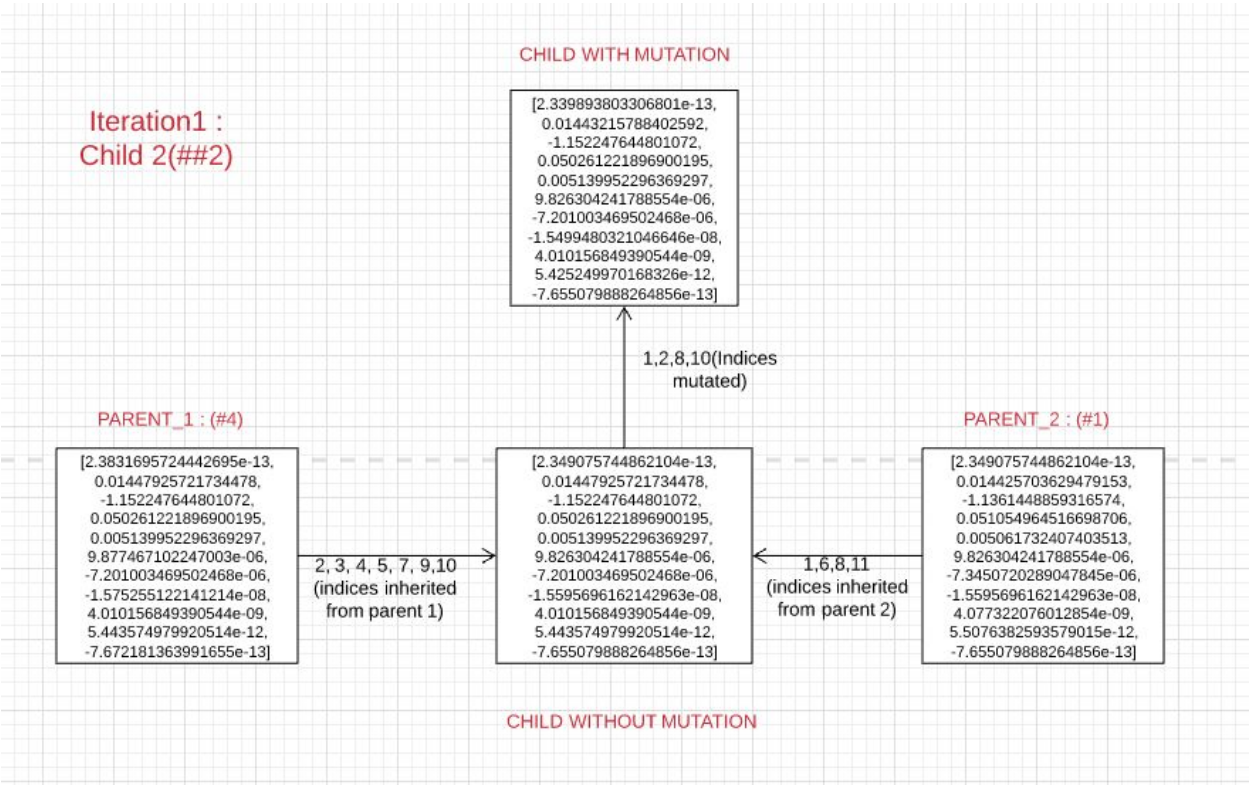
Iteration 1:



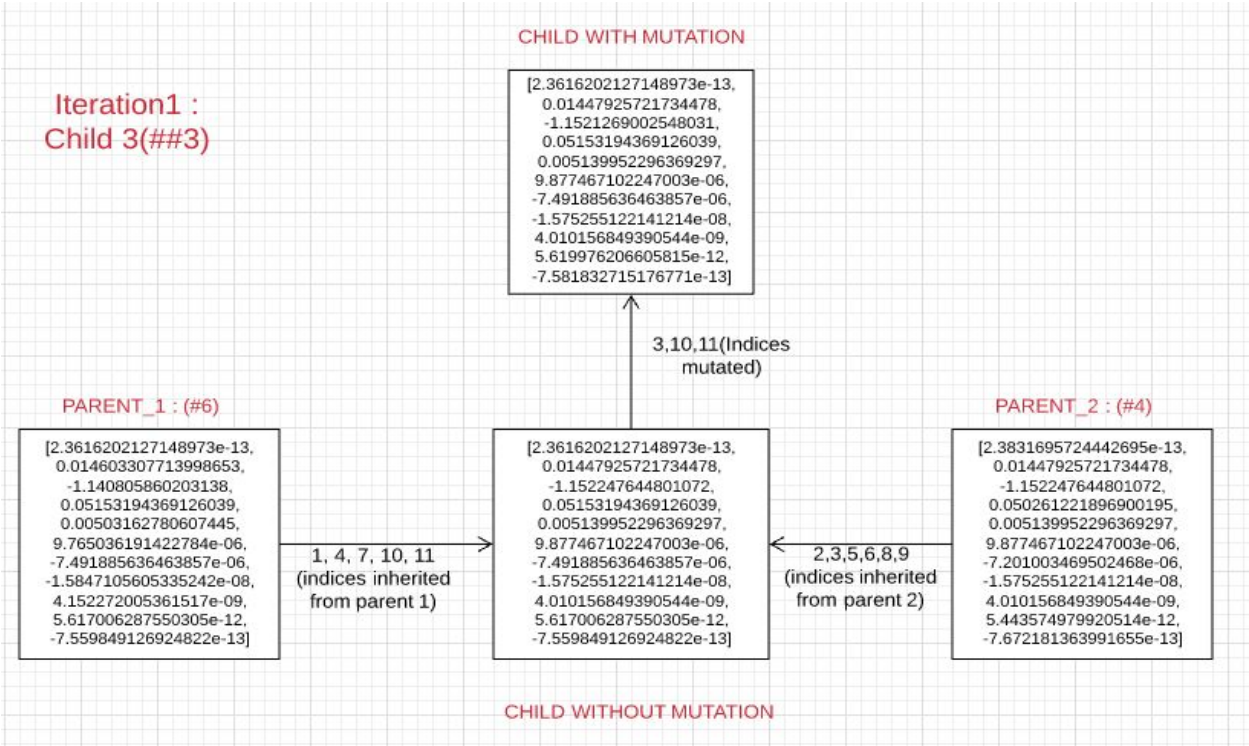
- **Child 1:**


CHILD WITH MUTATION
Iteration1 :
Child 1(##1)
[2.349075744862104e-13,
0.014443359103209517,
-1.1358137418080465,
0.051054964516698706,
0.005061732407403513,
9.877467102247003e-06,
-7.201003469502468e-06,
-1.5831622002372233e-08,
4.069508129594385e-09,
5.4012452494904466e-12,
-7.655079888264856e-13]
2,3,8,9,10(Indices mutated)
PARENT_1 : (#1)
[2.349075744862104e-13,
0.014425703629479153,
-1.1361448859316574,
0.051054964516698706,
0.005061732407403513,
9.826304241788554e-06,
-7.345072028904784 5e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.5076382593579015e-12,
-7.655079888264856e-13]
1, 3, 4, 5, 9, 11
(indices inherited
from parent 1)
[2.349075744862104e-13,
0.0144792572173447 8,
-1.1361448859316574,
0.051054964516698706,
0.005061732407403513,
9.877467102247003e-06,
-7.201003469502468e-06,
-1.575255122141214e-08,
4.077322076012854e-09,
5.443574979920514e-12,
-7.655079888264856e-13]
2,6,7,8,10
(indices inherited
from parent 2)
PARENT_2 : (#4)
[2.3831695724442695e-13,
0.0144792572173447 8,
-1.152247644801072,
0.050261221896900195,
0.005139952296369297,
9.877467102247003e-06,
-7.201003469502468e-06,
-1.575255122141214e-08,
4.010156849390544e-09,
5.443574979920514e-12,
-7.672181363991655e-13]
CHILD WITHOUT MUTATION

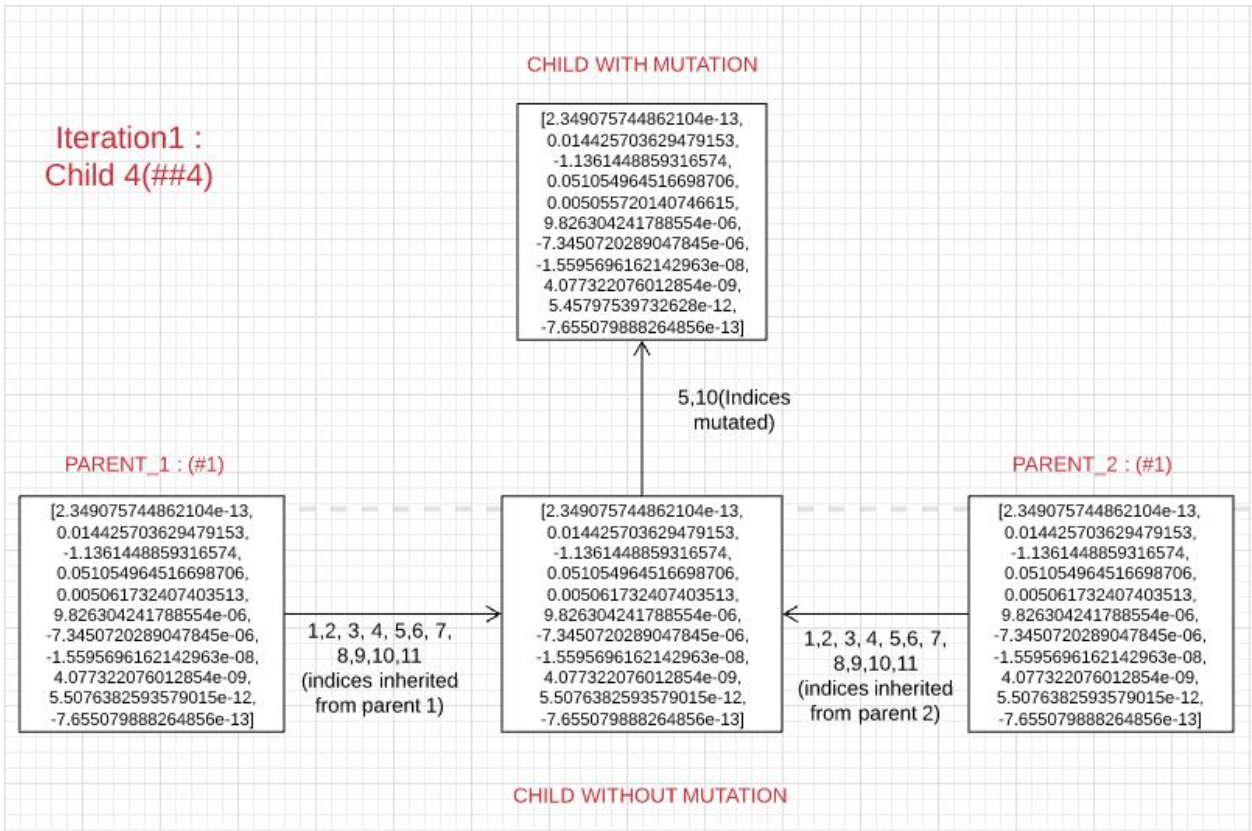
- **Child 2:**


CHILD WITH MUTATION
Iteration1 :
Child 2(##2)
[2.339893803306801e-13,
0.01443215788402592,
-1.152247644801072,
0.050261221896900195,
0.005139952296369297,
9.826304241788554e-06,
-7.201003469502468e-06,
-1.5499480321046646e-08,
4.010156849390544e-09,
5.425249970168326e-12,
-7.655079888264856e-13]
1,2,8,10(Indices mutated)
PARENT_1 : (#4)
[2.3831695724442695e-13,
0.0144792572173447 8,
-1.152247644801072,
0.050261221896900195,
0.005139952296369297,
9.877467102247003e-06,
-7.201003469502468e-06,
-1.575255122141214e-08,
4.010156849390544e-09,
5.443574979920514e-12,
-7.672181363991655e-13]
2, 3, 4, 5, 7, 9,10
(indices inherited
from parent 1)
[2.349075744862104e-13,
0.0144792572173447 8,
-1.152247644801072,
0.050261221896900195,
0.005139952296369297,
9.826304241788554e-06,
-7.201003469502468e-06,
-1.5595696162142963e-08,
4.010156849390544e-09,
5.443574979920514e-12,
-7.655079888264856e-13]
1,6,8,11
(indices inherited
from parent 2)
PARENT_2 : (#1)
[2.349075744862104e-13,
0.014425703629479153,
-1.1361448859316574,
0.051054964516698706,
0.005061732407403513,
9.826304241788554e-06,
-7.345072028904784 5e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.5076382593579015e-12,
-7.655079888264856e-13]
CHILD WITHOUT MUTATION

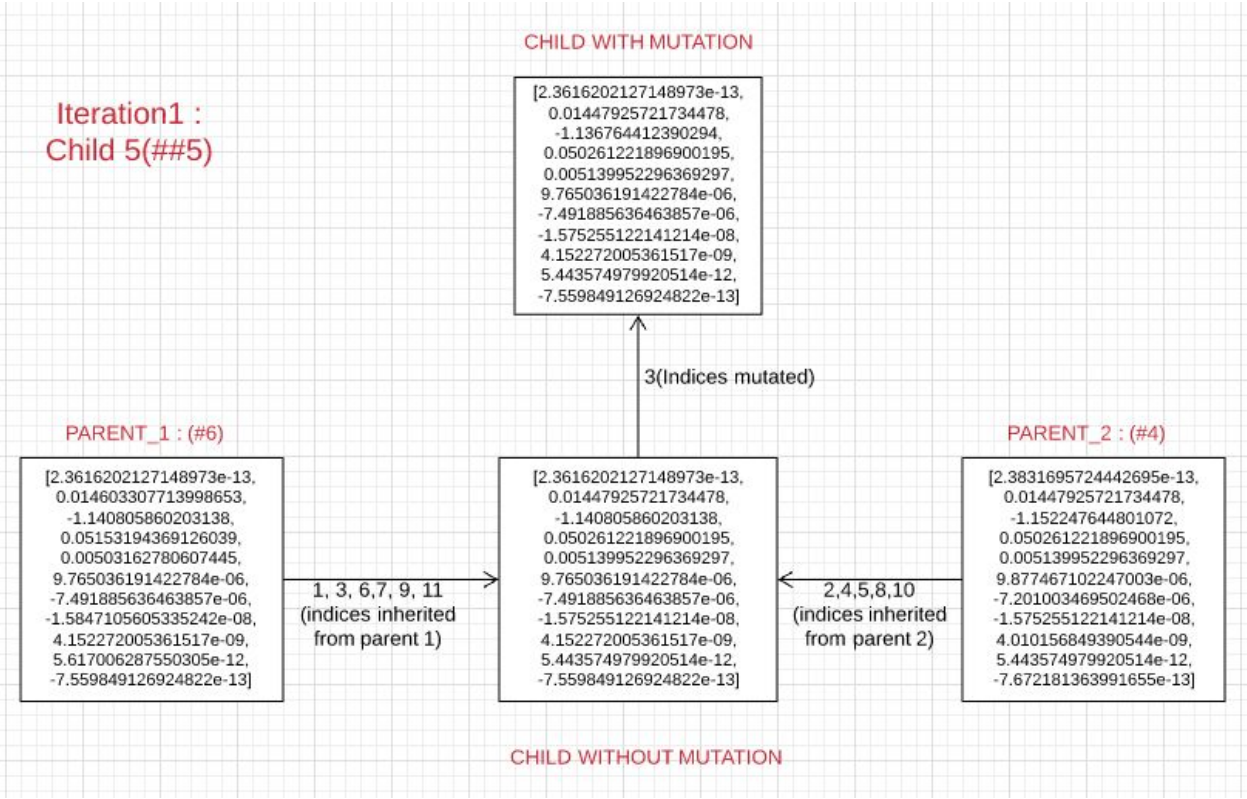
- **Child 3:**



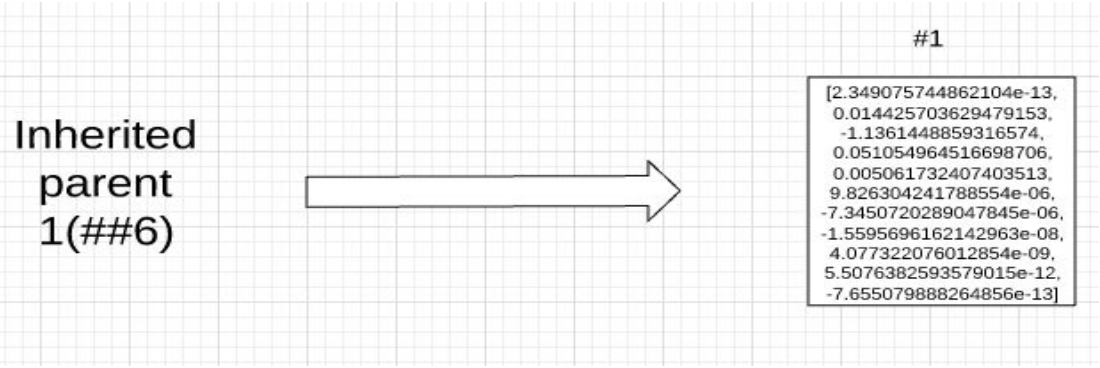
- **Child 4:**



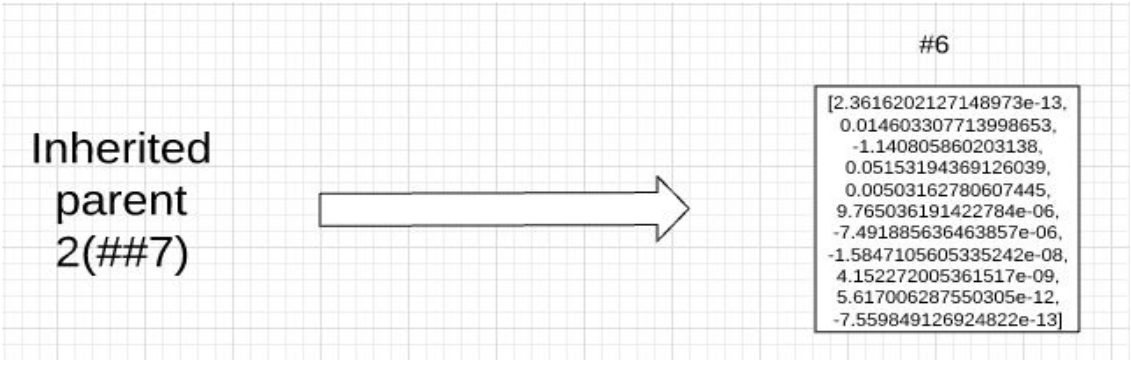
- **Child 5:**



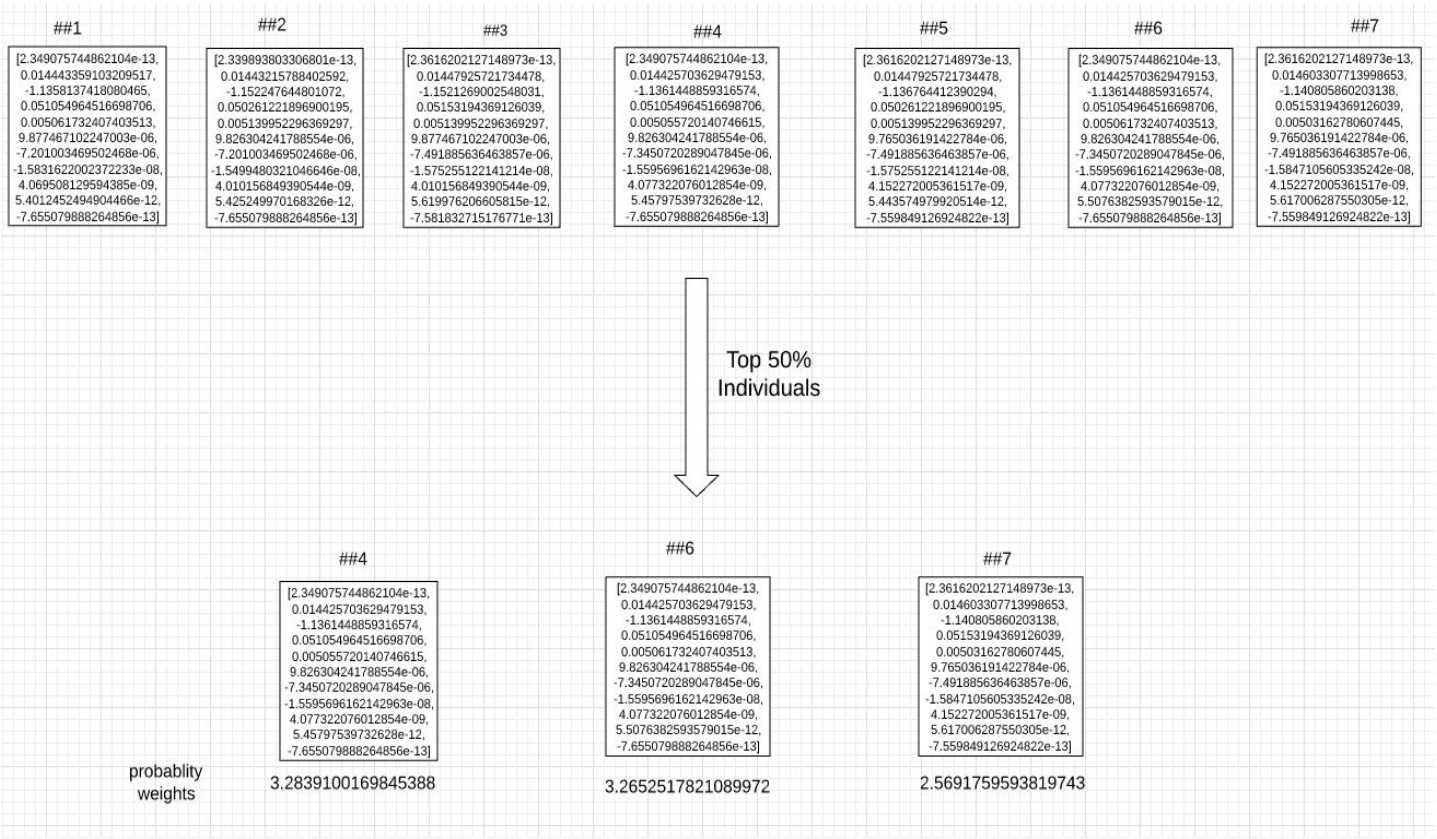
- **Child 6:**



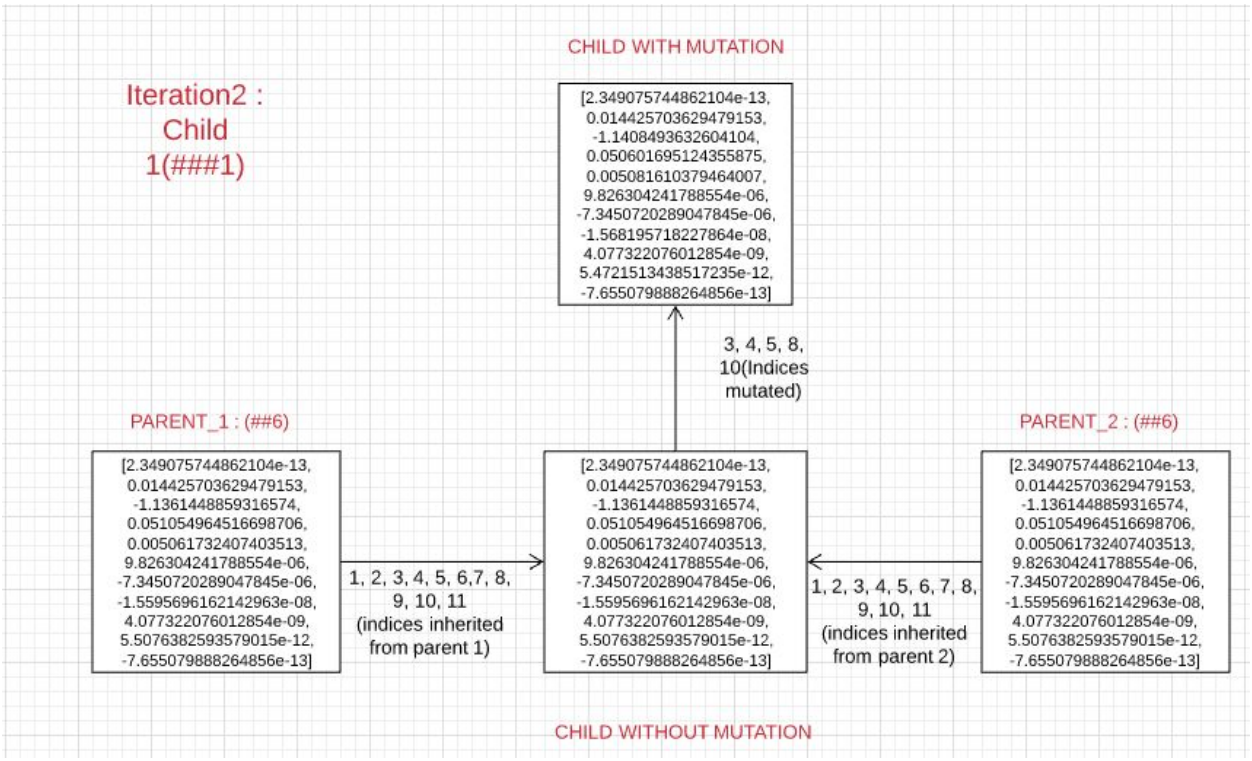
- **Child 7:**



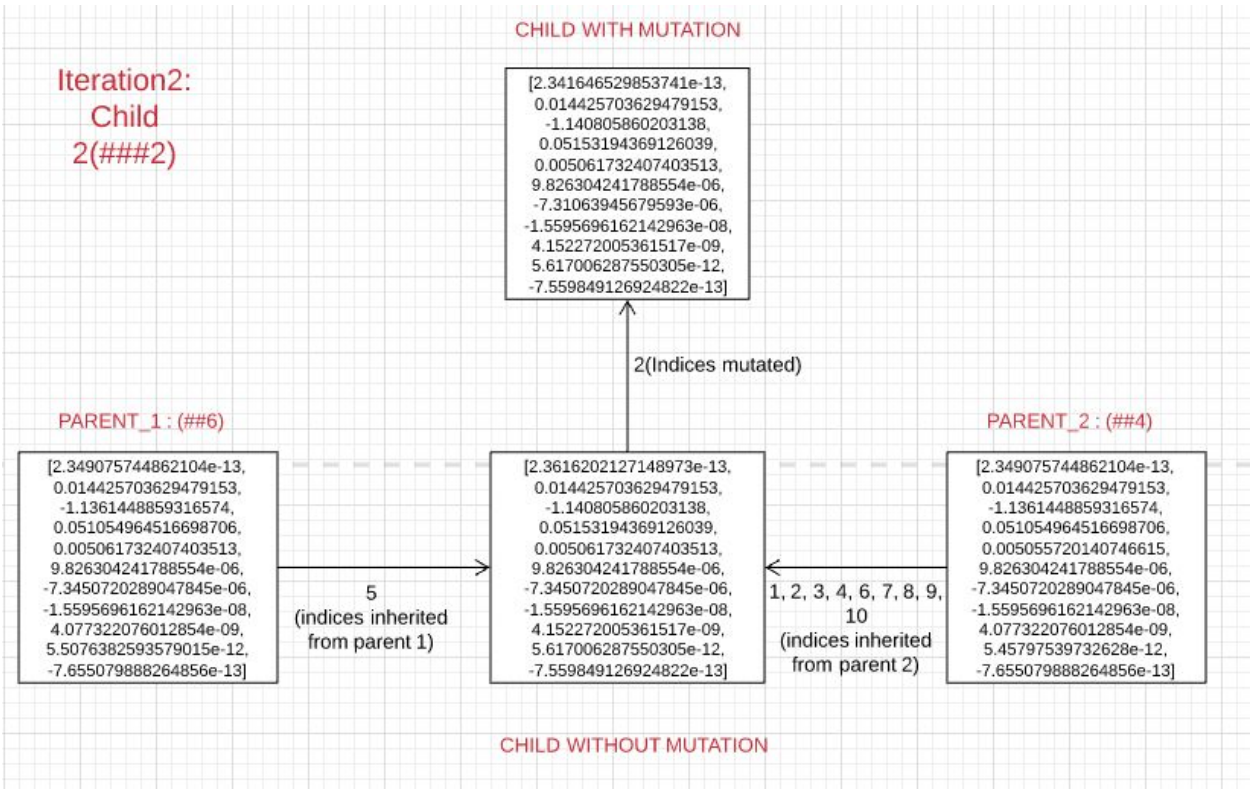
Iteration 2:



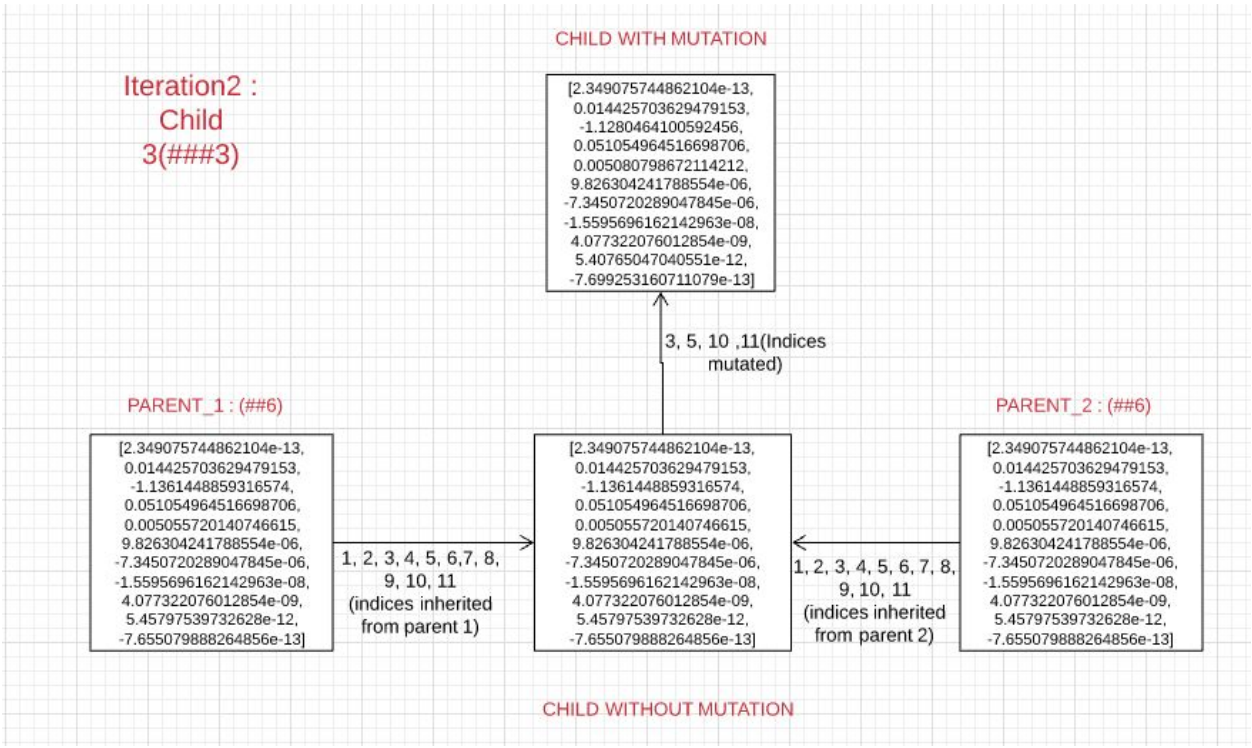
- **Child 1:**



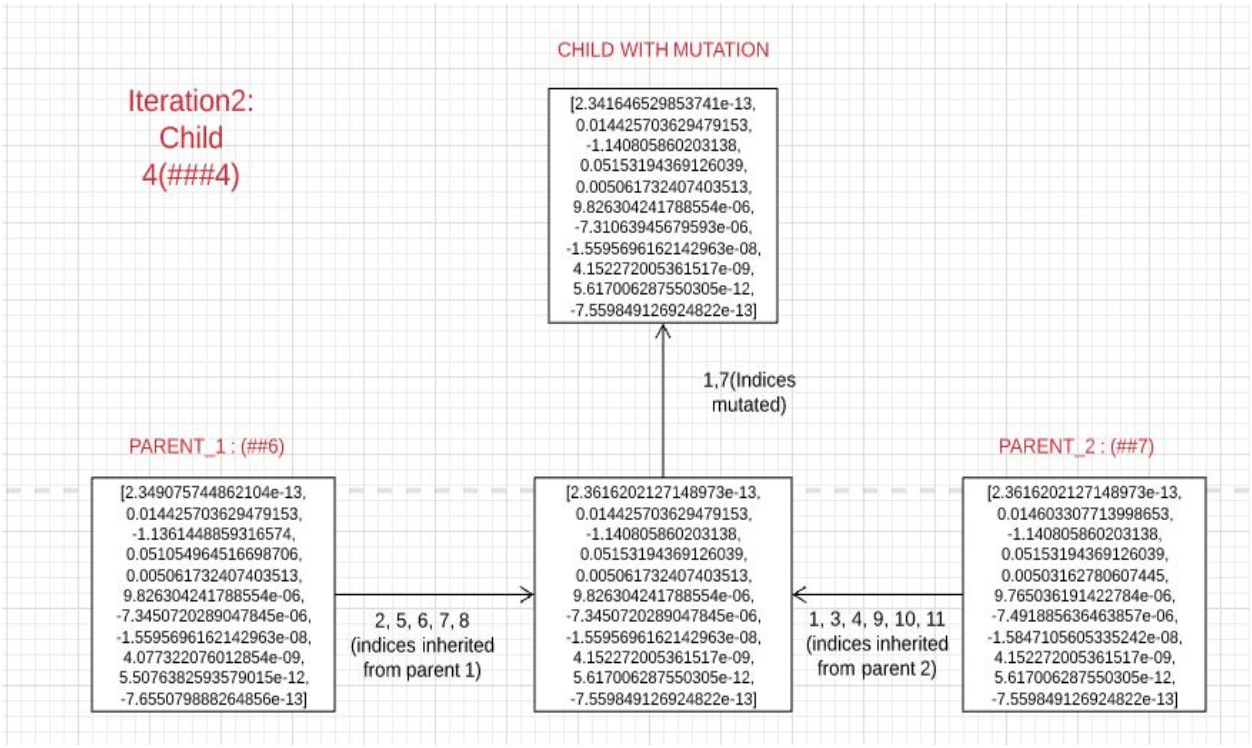
- **Child 2:**



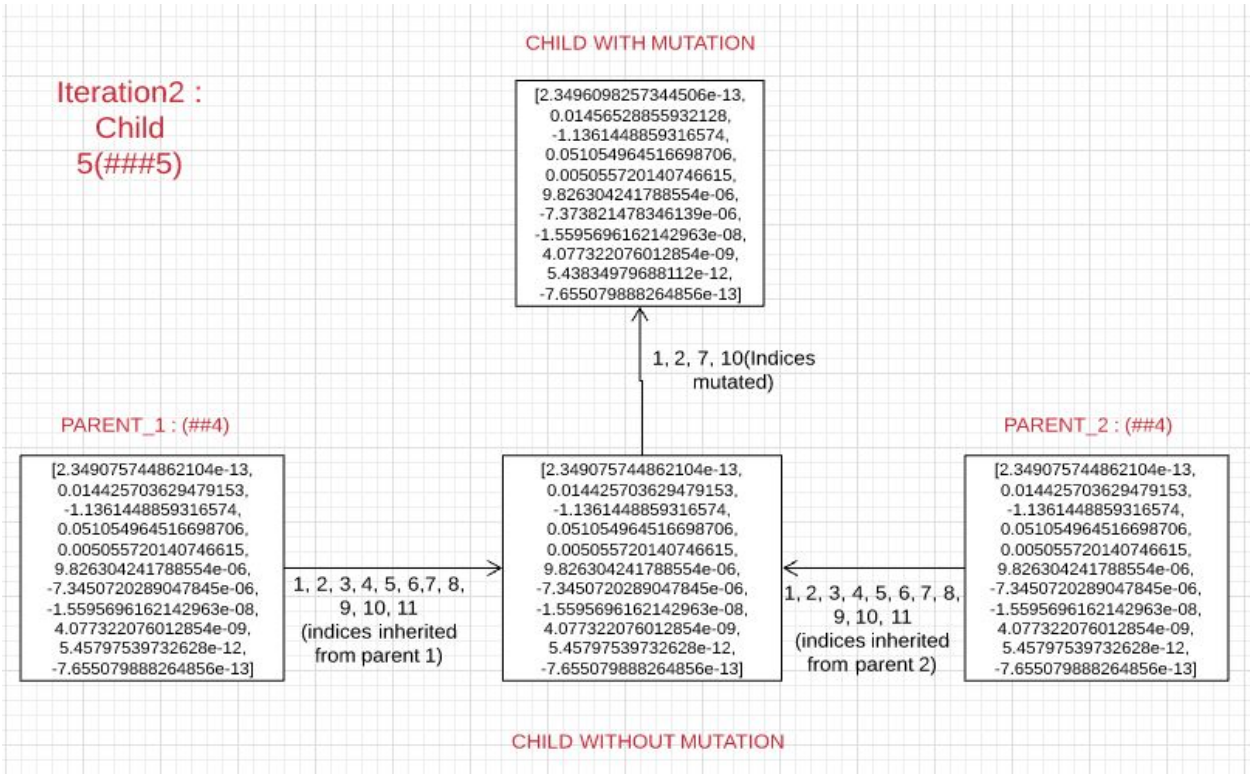
- **Child 3:**



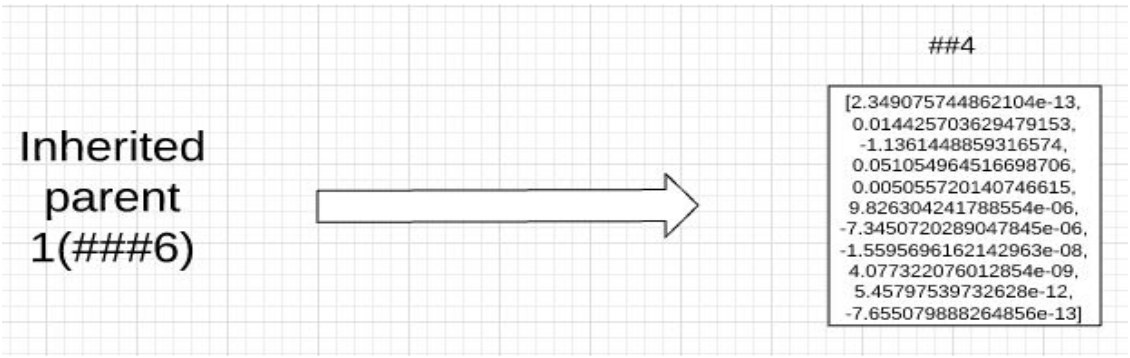
- **Child 4:**



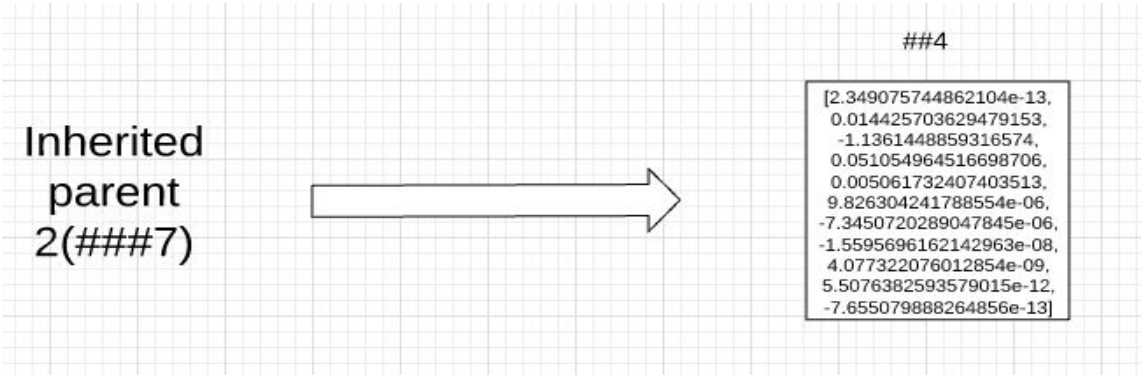
- **Child 5:**



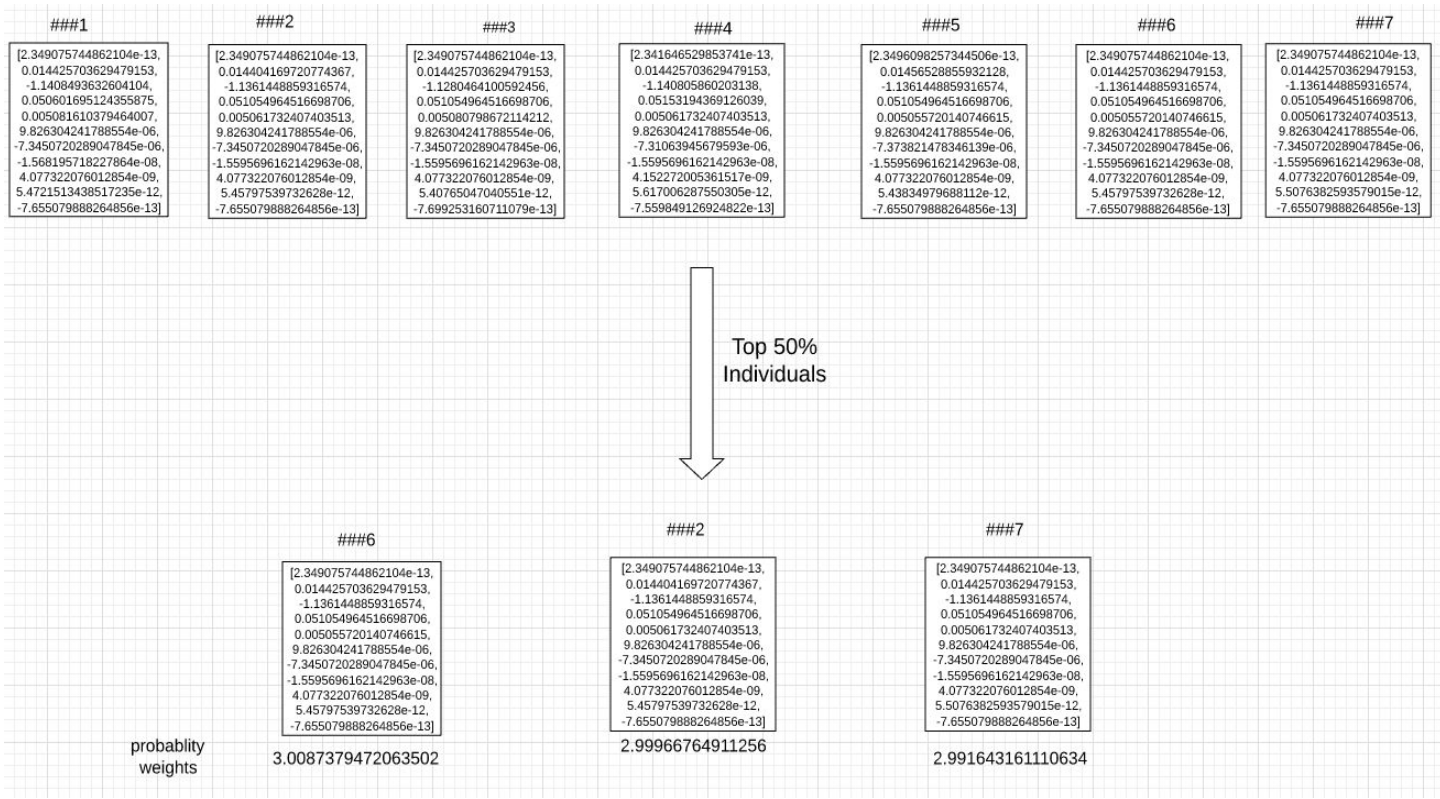
- **Child 6:**



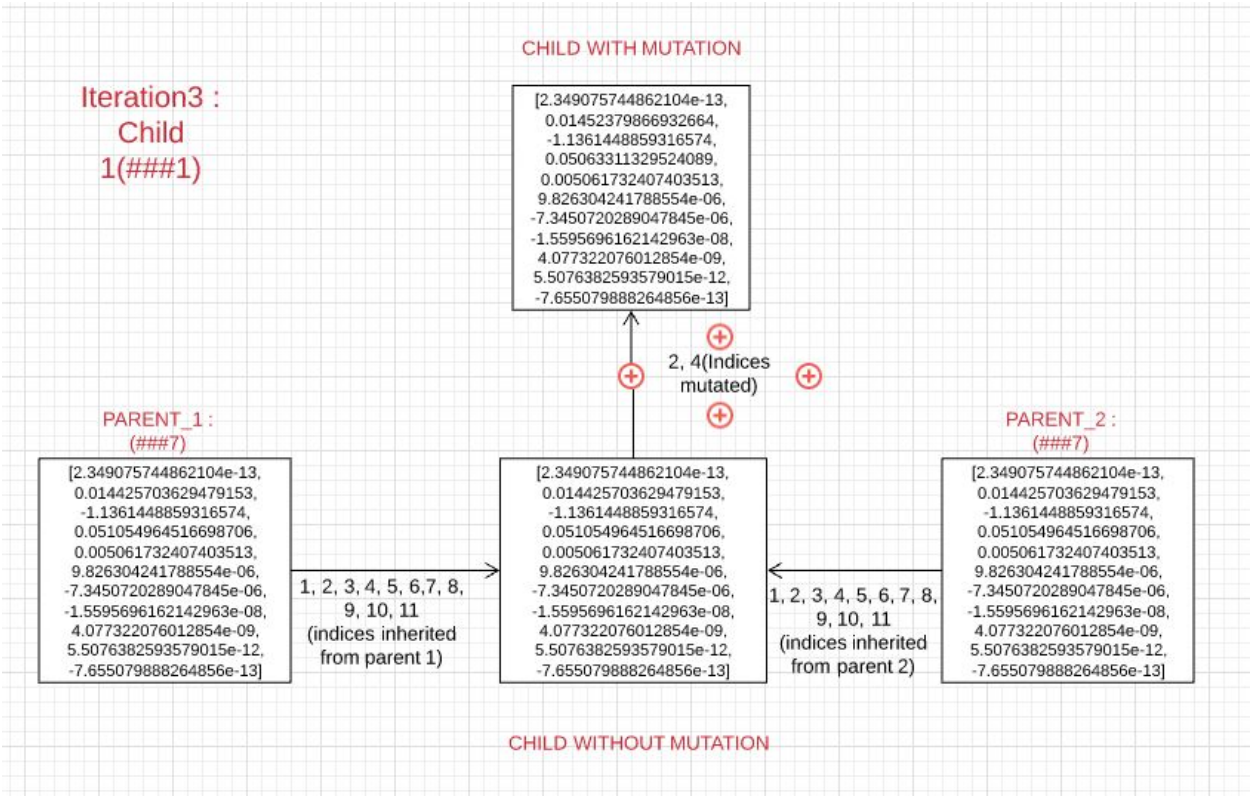
- **Child 7:**



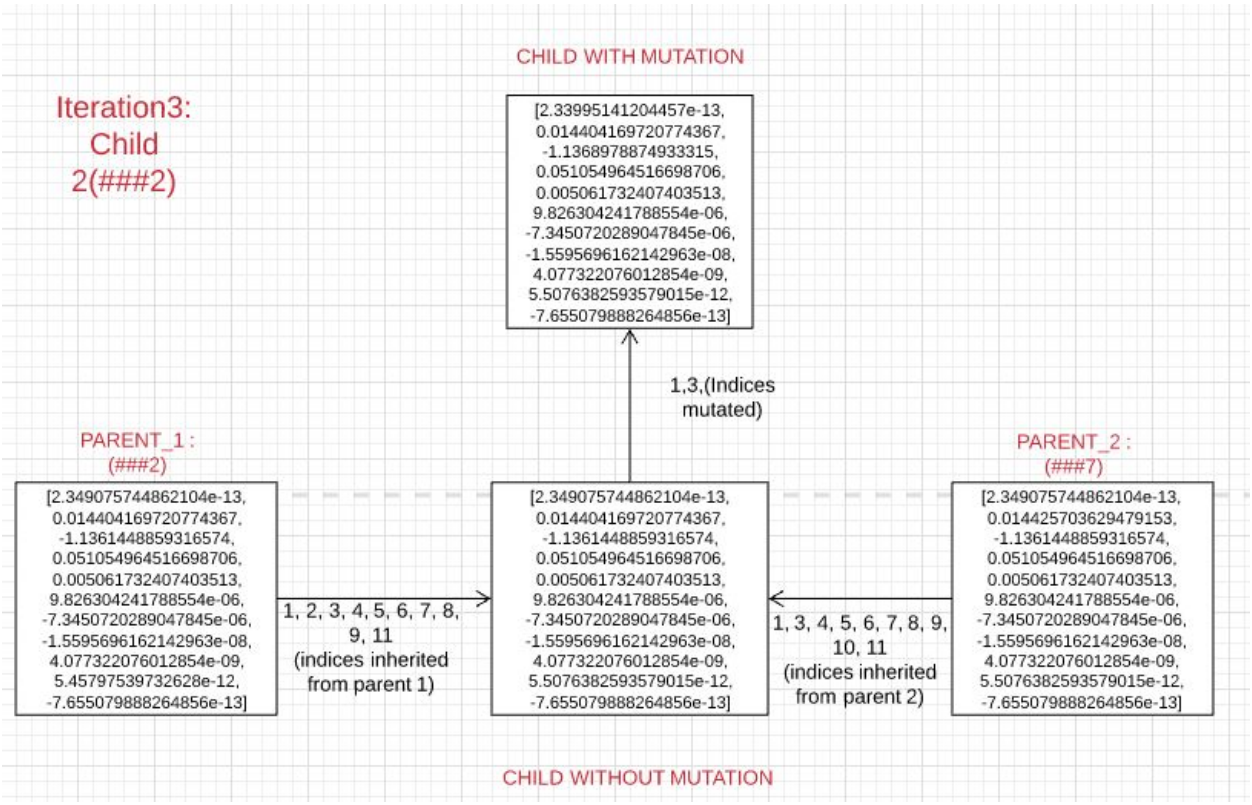
Iteration 3:



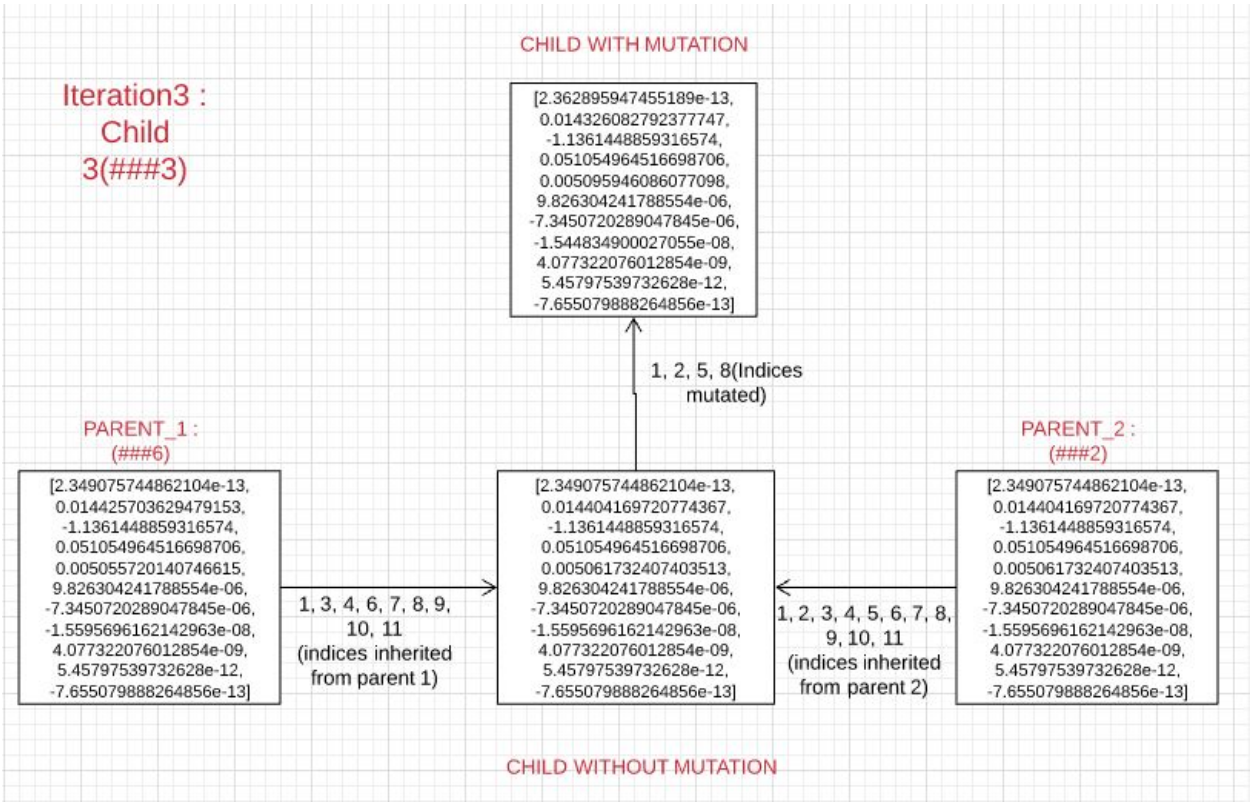
- **Child 1:**


CHILD WITH MUTATION
Iteration3 :
Child
1(###1)
[2.349075744862104e-13,
0.01452379866932664,
-1.1361448859316574,
0.05063311329524089,
0.005061732407403513,
9.826304241788554e-06,
-7.3450720289047845e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.5076382593579015e-12,
-7.655079888264856e-13]
2, 4(Indices
mutated)
PARENT_1 :
(###7)
PARENT_2 :
(###7)
[2.349075744862104e-13,
0.01442570362947913,
-1.1361448859316574,
0.05105496451669870,
0.005061732407403513,
9.826304241788554e-06,
-7.3450720289047845e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.5076382593579015e-12,
-7.655079888264856e-13]
1, 2, 3, 4, 5, 6,7, 8,
9, 10, 11
(indices inherited
from parent 1)
1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11
(indices inherited
from parent 2)
CHILD WITHOUT MUTATION

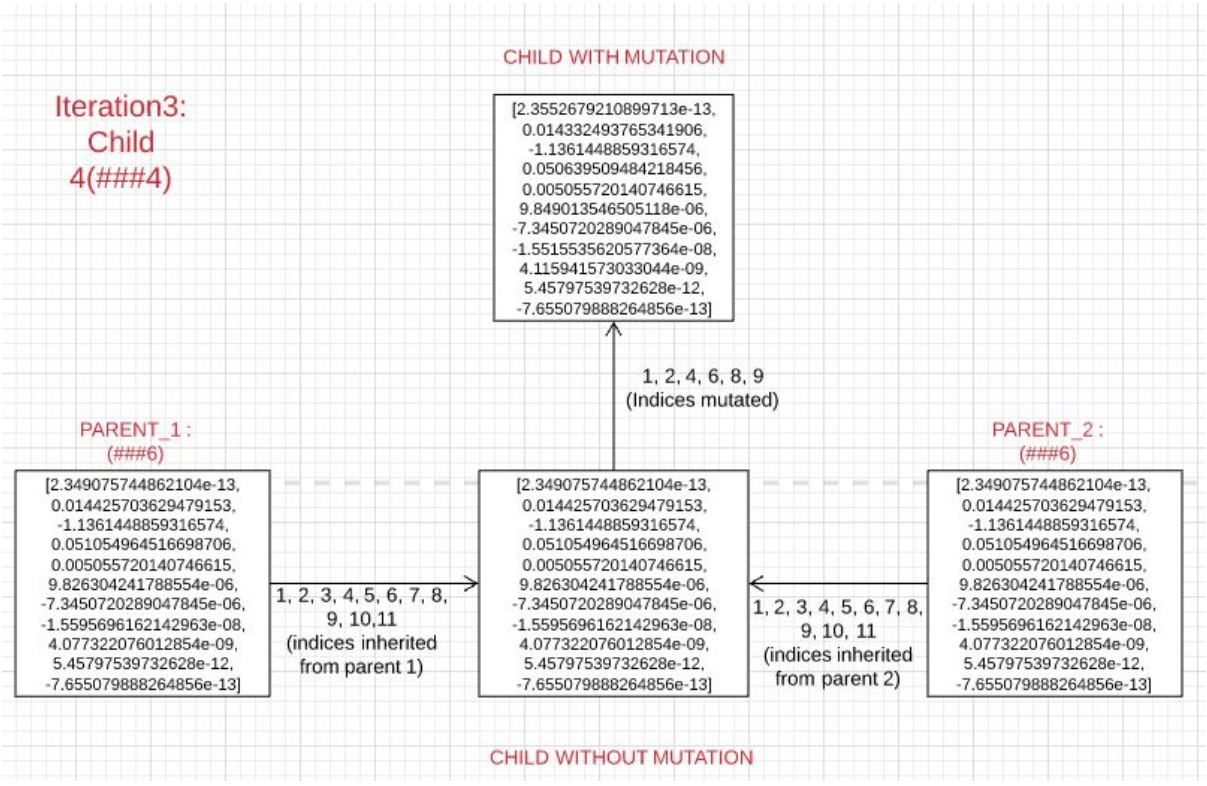
- **Child 2:**


CHILD WITH MUTATION
Iteration3:
Child
2(###2)
[2.33995141204457e-13,
0.014404169720774367,
-1.1368978874933315,
0.051054964516698706,
0.005061732407403513,
9.826304241788554e-06,
-7.3450720289047845e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.5076382593579015e-12,
-7.655079888264856e-13]
1,3,(Indices
mutated)
PARENT_1 :
(###2)
PARENT_2 :
(###7)
[2.349075744862104e-13,
0.014404169720774367,
-1.1361448859316574,
0.051054964516698706,
0.005061732407403513,
9.826304241788554e-06,
-7.3450720289047845e-06,
-1.5595696162142963e-08,
4.077322076012854e-09,
5.45797539732628e-12,
-7.655079888264856e-13]
1, 2, 3, 4, 5, 6, 7, 8,
9, 11
(indices inherited
from parent 1)
1, 3, 4, 5, 6, 7, 8, 9,
10, 11
(indices inherited
from parent 2)
CHILD WITHOUT MUTATION

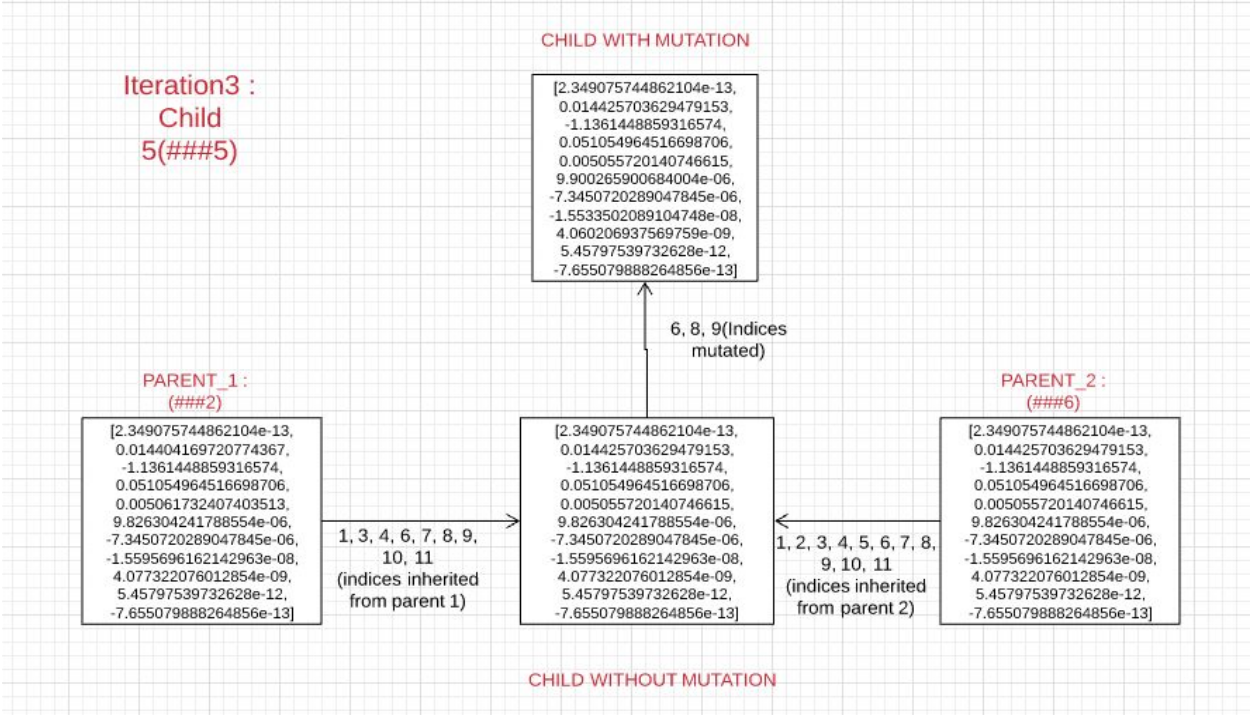
- **Child 3:**



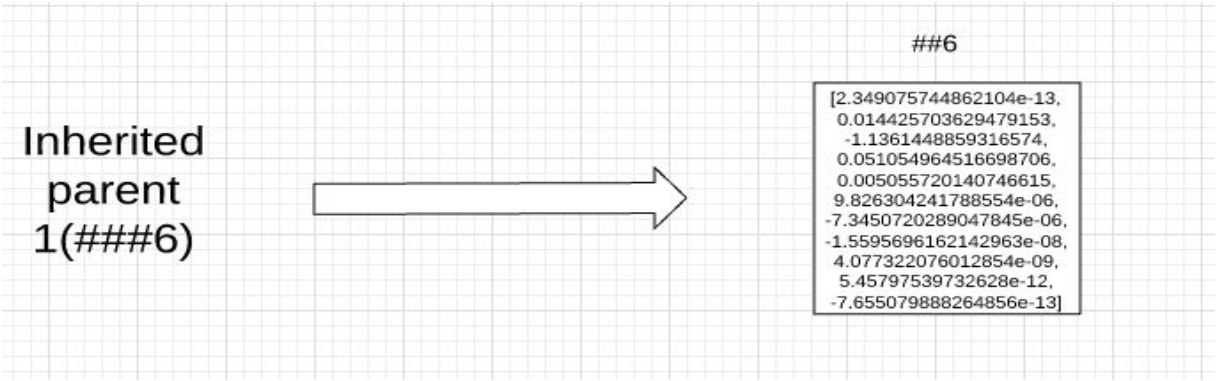
- **Child 4:**



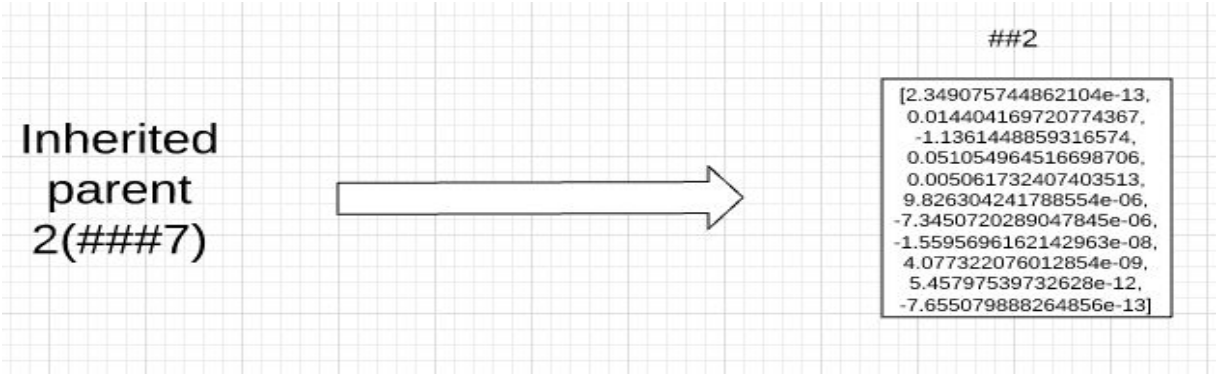
- **Child 5:**



- **Child 6:**



- **Child 7:**



Trace Specs

The trace of 10 iterations have been redirected in a file. Submitted the file with the name **trace.txt**.

References Taken

1. GeeksForGeeks
2. Artificial Intelligence, A Modern Approach (by Stuart Russell and Peter Norvig)