MDL Project Report

Summary

A genetic algorithm is a machine learning algorithm that employs natural selection and other evolutionary techniques to improve its model. The problem statement we had was: We were given the coefficients of a linear equation in 11 variables. The coefficients corresponded to an overfit model and our task is to use genetic algorithms to steer this overfit model to give us a set of better-fit ones. The training and validation sets of data were not explicitly given, rather a fixed number of queries per day was given and one could query the training and validation errors for their model.

Since it was to be a genetic algorithm, there were to be some mandatory things like mutation, crossover, and elitism to be involved. But the interesting part was to dilute the overfitted model. Let's see a summary of what we did to solve the above problem.

Start: The algorithm starts by taking input, the overfit vector of coefficients, to operate upon. This single overfit vector is then mutated (we tried various ways to mutate this) to give us a population of vectors which apparently become the first generation of our algorithm.

We tried out several ways to mutate the overfit array. The most commonly used is to multiply about 4-6 coefficients with random real numbers between 0.8 to 1.2. Some other tricks we tried were to initialize all children by 0.0 and then add a few mutated genes to each of them, but the other coefficients were left to 0. Yet another method we tried out was to perform asexual reproduction of the overfit array, i.e., the overfit vector was made to mate with itself.

Also, note that we altered the population size itself to balance the minimum number of requests and the diversity we received. Evidently and obviously, diversity meant a larger population whereas it also meant large numbers of requests had to be spent on each generation meaning we could try out fewer heuristic changes in our algorithm per day. We found 20 to be the best population size but we often varied this and we were benefitted as well.

Iterative logic:

1. The errors of each individual of this population are queried and each individual is ranked on the basis of a cocktail of this set of errors. The cocktail basically means a linear combination of both errors (training and validation). The linear combination was of the form "T_error + validaiton_factor * V_error" and we varied validation_factor from 0.1 to 5 to see a range of results which we have explained in detail further in the report.
2. This leaderboard is used to perform privilege-based mating to produce the next generation. Top K individuals can only mate to produce the offsprings.
   The number K was decided on the basis of the generation that we expected to get and was typically in the 40% to 60% range. Please note that we varied the K in realtime to get the best results.

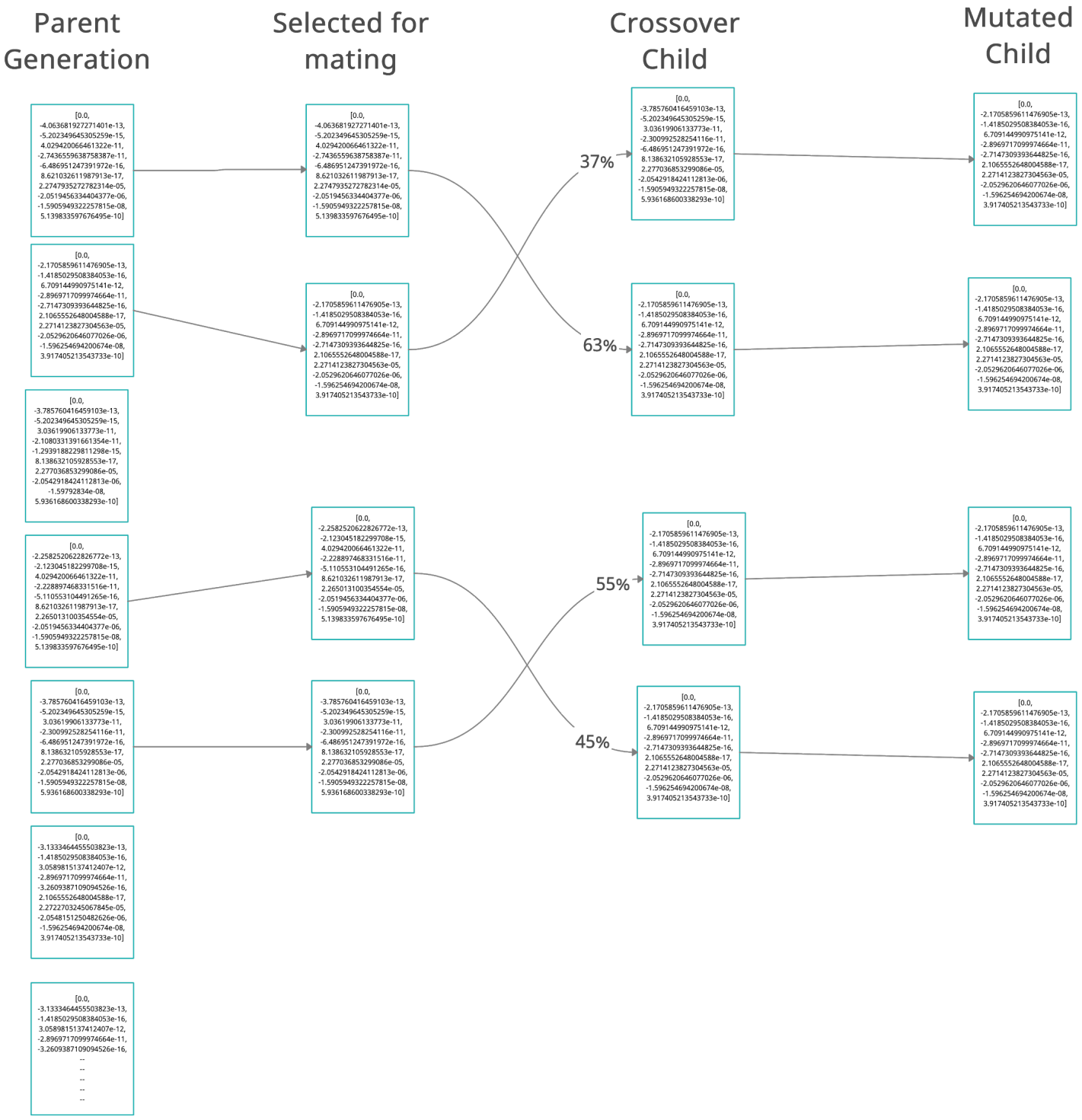
3. After having mutated to give us the next generation, we go back to step 1 after printing the necessary details of this particular iteration.
   The logarithm of each iteration served as a crucial tool to analyze the iterations and understand and ideate the changes to further improve our algorithm. We have mentioned our changes in great detail below.

End: The algorithm stops when the convergence criteria is met or if the stipulated number of iterations have been performed.
The convergence criteria we used were mostly based on how the generation aged. We looked at the leaderboard to compare the present generation to the previous one and the threshold was to at least keep the first 6-7 vectors intact from the best of the previous generations. When we saw that the numbers went down significantly, that's when the algorithm stopped and we'd say that we were diverging by the time we stopped.
Other methods were relatively simple such as limiting the number of generations or stopping when the best vector reached a particular error threshold.

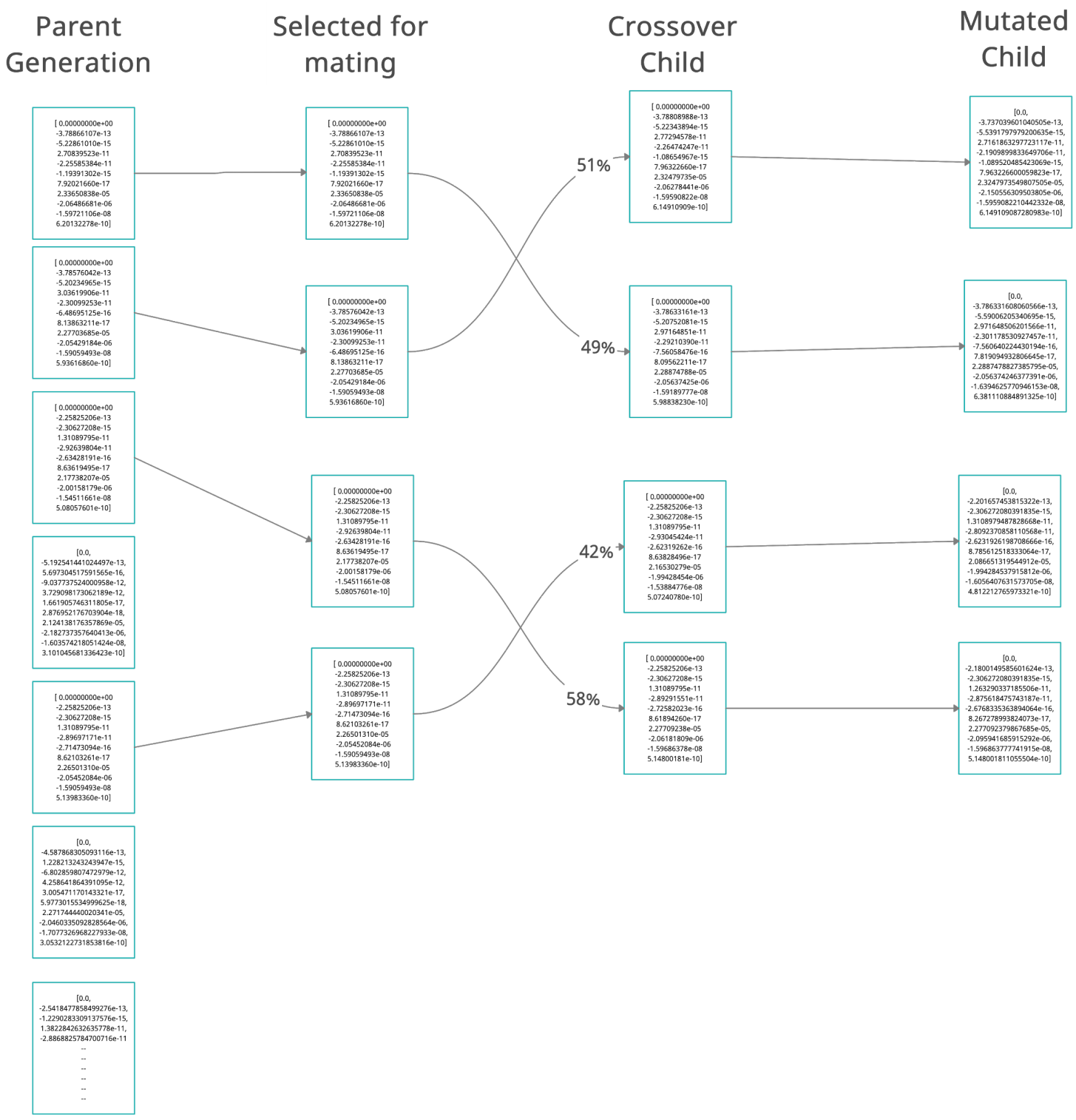
Iteration Diagrams :-

The initial population so written in Generations 1,2,3 are sorted according to thier fitness value(lesser the value higher the priority) which is calculated through the fitness function we used(refer **fitness function for more details**).
Selection of parents for mating is done by randomly choosing top 10-12 vectors of initial population.
This way we were able to maintain the diversity in population too.

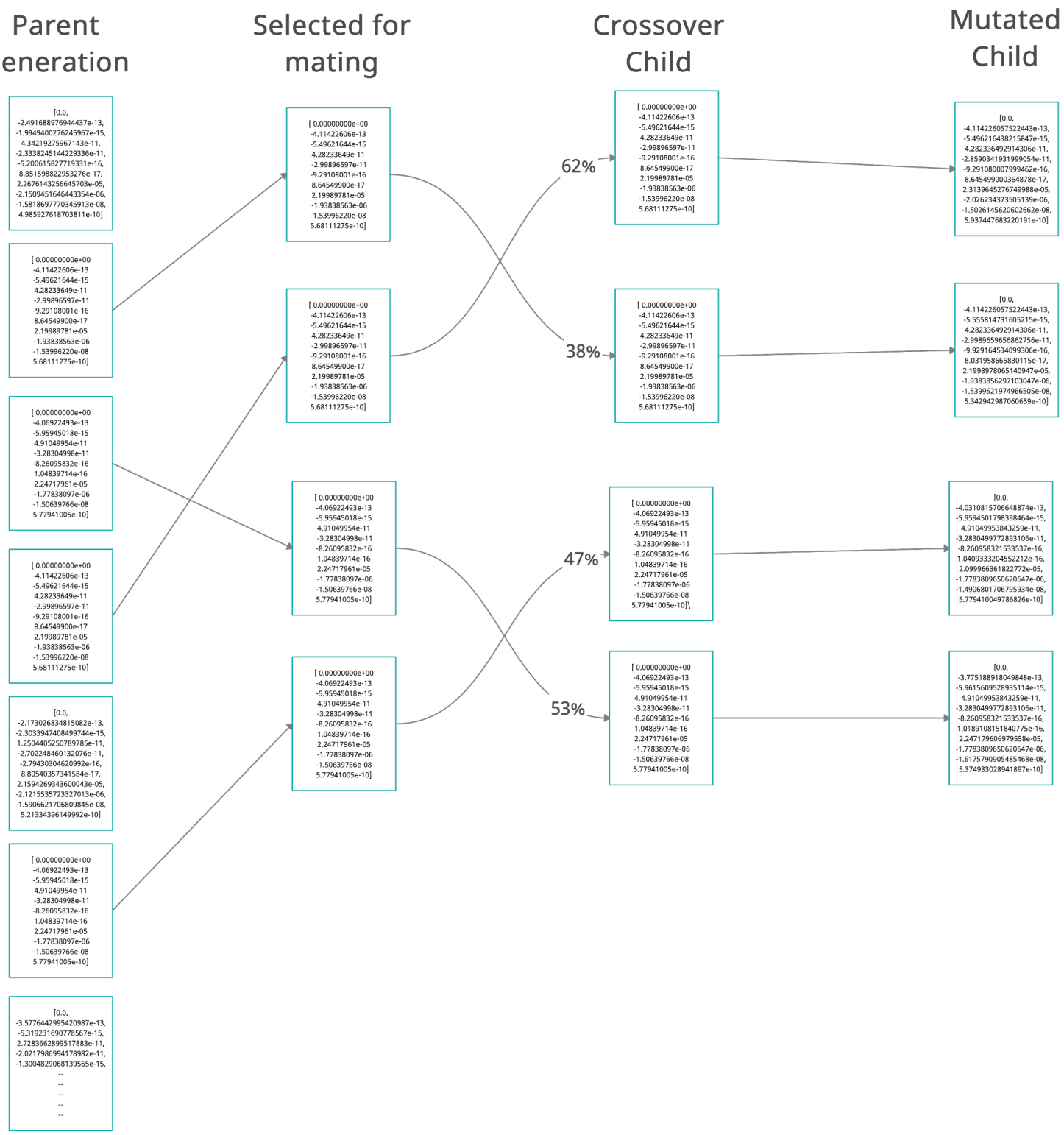
Generation 1



Generation 2



Generation 3



Fitness function

In the fitness function, get_errors requests are sent to obtain the train error and validation error for every vector in the population. The fitness corresponding to that vector is calculated as absolute value of 'Train Error + Validation Error*validation_factor'.

```
def cal_fitness(self):
    fitness = abs(valid_factor*self.validation_error +
self.training_error)

    return fitness
```

We changed the value of the “validation_factor” from time to time, depending on what our train and validation errors were for that population. This helped us achieve a balance between the train and validation error which were very skewed in the overfit vector. We kept changing between the following 2 functions according to the requirement of the population (reason for each function mentioned below)

Validation_factor > 1

We used validation_factor (1, 2) to **get rid of the overfitting**. The initial vector had a very low train error and more validation error. With our fitness function, we associated less weight to the training error and more to validation error, forcing validation error to reduce yet not allowing train error to shoot up. We also changed it to values between **2 and 3** in the middle to get rid of the overfitting faster. However, values between **(1.1, 1.5)** worked best as it did not cause train error to rise up suddenly, unlike its lower values.

Validation_factor <= 1

This was done when validation error became too less than training error.(which was not intended as we were asked to reduce both training error and validation error simultaneously. We kept values between **(0.6, 0.8)** to make both errors become equal. After the difference
b/w the two errors became insignificant we kept validation_factor in range **(0.95,1.1)** to reduce both errors equally.

Crossover

Single point crossover

Initially, we implemented a simple single-point crossover where the first parent was copied till a certain index, and the remaining was copied from the second parent.

However, this offered very little variation as the genes were copied directly from either parent. We read research papers and found out about a better technique (described below) that we finally used.

Simulated-Binary-Crossover

The entire idea behind simulated binary crossover is to generate two children from two parents, satisfying the following equation. All the while, being able to control the variation between the parents and children using the distribution index value.

$$\frac{x_1^{new} + x_2^{new}}{2} = \frac{x_1 + x_2}{2}$$

The crossover is done by choosing a random number in the range `[0, 1)`. The distribution index is assigned its value and then beta is calculated as follows:

$$\beta = \begin{cases} (2u)^{\frac{1}{\eta_c+1}}, & \text{if } u \leq 0.5 \\ \left(\frac{1}{2(1-u)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise} \end{cases}$$

> Distribution index that determines how far children go from parents. The greater its value the closer the children are to parents.

The distribution index is a value between [2, 5] and the offsprings are calculated as follows:

$$x_1^{new} = 0.5\left[(1+\beta)x_1 + (1-\beta)x_2\right]$$
$$x_2^{new} = 0.5\left[(1-\beta)x_1 + (1+\beta)x_2\right]$$

```
# here self is the parent1(written as a class function)
def mate(self, parent2):
    child1 = np.empty(11)
```

```
    child2 = np.empty(11)

    u = random.random()
    if (u < 0.5):
       beta = (2 * u)**((dist_index + 1)**-1)
    else:
       beta = ((2*(1-u))**-1)**((dist_index + 1)**-1)

    parent1 = np.array(self.chromosome)
    parent2 = np.array(parent2.chromosome)

    child1 = 0.5*((1 + beta) * parent1 + (1 - beta) * parent2)
    child2 = 0.5*((1 - beta) * parent1 + (1 + beta) * parent2)
```

We varied the **Distributed Index** value depending on the population.

Mutation

Our mutations are probabilistic in nature. For the vector, at every index a mutation is decided to be performed with a probability of p.(we varied p between **4/11 to 9/11** according to variations we need in our population and which best suits for a particular population)

We scale the value at an index by randomly choosing a value between (1-delta, 1+delta), where delta is taken in range from **(0.05,1.15)**. Iff the value after scaling is within the valid (-10, 10) range. The following code does that:

```
# Mutation(here mutation_prob is kept b/w 4-9)
 for i in range(num_of_features):
     prob = random.randint(0, 10)
     if prob < mutation_prob:
        vary = 1 + random.uniform(-delta, delta)
        rem = child1[i]*vary
        if abs(rem) <= 10:
           child1[i] = rem

      prob = random.randint(0, 10)
      if prob < mutation_prob:
         vary = 1 + random.uniform(-delta, delta)
         rem = child2[i]*vary
         if abs(rem) <= 10:
            child2[i] = rem
```

- Initially we chose value b/w **(-10,10)** for mutations but it hadn't flourished good results.Later we realize that the overfit vector is close to good results but has just

overfit on the training data. Thus, tweaking it slightly would give improved results!

- We later kept delta for the purpose of mutation.We changed these mutations as per the trend of the previous populations. If we observe the errors were not reducing significantly over generations, we increased the delta between (0.1, 0.15) and even (0.3,1) and other variations in the middle. We would experimentally observe which helped us get out of a local minima.(we got stuck one time at log(validation_err)= 11.1 and log(training_err)=11.3,..after good mutations we landed at values whose logarithm were 10.82/10.6)

- Sometimes, we even decreased the mutations further, to reach a finer granularity of our genes. We did this when we were confident we had good vectors that needed more fine tuning. We set the delta between (0.025, 0.075) for them.

Hyperparameters

Population size

The POPULATION_SIZE parameter is set to **20**. We initially started out with POPULATION_SIZE = 8-10. But soon we realized that our population is lacking diversity. So we kept it to 20 and have gone over most of the days with size 20 only to have a decent diversity maintained.

Also, we raised it to 30 for having more varied results but that hadn't produced loud results.

So we went with a population size of 20 over our whole assignment.

Mating pool size

The MATING_POOL_SIZE variable is changed **between values of 6 and 15**. We sort the parents by the fitness value and choose the top X (where X varies between 6 to 15 as we set it) that are selected for the mating pool.

- In case we get lucky or we observe only a few vectors of the population have a good fitness value, we decrease the mating pool size so that we can limit our children to be formed from these high fitness chromosomes. Also, when we observe that our GA is performing well and do not want unnecessary variations, we limit the mating pool size.
- When we find our vectors to be stagnating, we increase the mating pool size so that more variations are included in the population as the set of possible parents increases.
- Most often when the population was good to go for the next generation we kept the mating pool size to be 12 and got good results.

Number of parents passed down to new generation(aka Elitism)

We kept elitism around **10% to 20% of the initial population**.

- We kept the value small when we were just starting out and were reliant on more variations in the children to get out of the overfit. We did not want to forcefully bring down more parents as that would waste a considerable size of the new population.

- When we were unsure of how our mutations and crossover were performing or when we would change their parameters, we would increase this variable to 20. We did this so that even if things go wrong, our good vectors are still retained in the new generations. This would save us the labor of manually deleting generations in case things go wrong as if there is no improvement by our changes, the best parents from above generations would still be retained and used for mating.

Distribution index (Crossover point)

This parameter was applied in the Simulated Binary Crossover. It determines how far children go from parents. The greater its value the closer the children are to parents. It varies from 2 to 5. We changed the Distribution Index value depending on our needs. When we felt our vectors were stagnating and needed variation, we changed the value to **2, so that the children would have significant variations from their parents**. When we saw the errors decreasing steadily, we kept the index as **5 so that children would be similar to the parent population, and not too far away**.

Mutation Range

We varied our mutation range drastically throughout the assignment.

- We made the variation as little as, delta ranging in(0.01,0.05) for when we had to fine-tune our vectors. We did this when we were confident we had a good vector and excessive mutations were not helping. So we tried mall variations, to get its best features.

- When our vectors would stagnate and reach a local minima - we would mutate extensively to get out of the minima. The delta varied anywhere from (0.1, 0.15) to (0.2, 0.5).

- Exact details as to how we did this can be found in the **Mutation** section.

Convergence Criteria

The convergence criteria mainly consisted of two conditions:

- Deteriorating Generation
- Limit on number of generations

To save a number of requests, we periodically checked the generations (we kept the number of generations to 3 to analyze the population after applying the modifications and we continued from our last left population if generation seems to produce loud results) and made sure that the successive generations didn't go too bad. A convergence point is visible when the generation starts to deteriorate in successive iterations and the population of good vectors decreases and errors start to rise again. It did happen from time to time that we voilated this convergence criteria and we got improved results, but we stopped when there was no improvement even after a lot of iterations.

Heuristics

- **Initial vector**: Firstly we kept values of initial vector randomly chosen from (-10,10).But we realized that this was actually not working as errors seemed to be quite huge.

  After that we also tried by taking all values of the initial vector to be 0.0.This worked far better than our last approach but still it also could not produce the results we were expecting.

  So later we tried with an overfit array(given with the assignment) to reduce our errors and it really produced astonishing results.After reaching a sufficiently good vector we again started off our whole process with this new vector as initial vector in motivation of getting dreaming results, and viola! This actually worked, and we continued this process till the clock striked the deadline.

- **Probabilistic mutation**: Earlier we were mutating on one index only. But we changed our code to mutate each index with a probability of p (p lies in range of 3/11-8/11), this brought more variation in the genes and worked well for our populations.

- **Creating initial population from overfit array**: At first we blindly used the overfit array for our initial population but that led the overfit array to stay on top of our population for many generations because of elitism. Later we generated a multiplier(close to 1) which was multiplied with each index value of the overfit array.

  Here we applied a significantly important heuristics of generating 11 multipliers according to the value at that position. *We noticed that values at indices 7,8,9 were close and also large enough than other values in the vector*. So we need a multiplier viz. quite close to 1.For rest of the multipliers as the values are too low in magnitude we kept large multipliers for them.Here's the code:-

```
def create_gnome():
  '''
  create chromosome or string of genes
  '''
  new_individual = overfit_arr
  for index in range(num_of_features):
      vary = 0
```

```
        prob = random.randint(0, 10)
        if prob < mutation_prob:
            if (7<= index) and (index <= 9):
                vary = 1 + random.uniform(-0.005, 0.005)
            else:
                vary = random.uniform(0, 1.5)
            rem = overfit_arr[index]*vary

            if abs(rem) < 10:
                new_individual[index] = rem
            elif abs(new_individual[index]) >= 10:
                new_individual[index] = random.uniform(-0.01,0.01)
    return new_individual
```

- Variations in fitness function, mating pool size, population size are also heuristics that we applied as the algorithm and the code could not detect when these changes were required. We had to manually study our population and see the impact of these variations and accordingly modify them.

Minimized errors

Vector:- *[0.0, -8.207201231701481e-13, -3.4646677272210014e-13, 1.2539007889352678e-11, -1.269892688156442e-10, -3.1361067404505953e-16, 2.35725161844418e-16, 2.4581226234021372e-05, -1.5368822606977392e-06, -1.4379608427031595e-08, 6.910595571608471e-10]*

Training Error:- *41527082786.94388*

(logarithm to the base 10 value of Training Error is 10.61833142366871)

Validation Error:- *32262527107.04966*

(logarithm to the base 10 value of Validation Error is 10.508698382447303)

We feel that this is the best vector we have since both training and validation errors are close to each other as well as lesser than most of our other models. At other times, if we were able to decrease our validation error, then it meant overfitting the validation set therefore the training error rose significantly and vice versa. Independently, the validation error came down to as low as 10^(10.29) and training error to 10^(10.35).

The model should perform good on unseen data as well since it has very little difference in performance in the training and validation sets. The model was generated using generalized methods and the errors show that there was no overfitting. Further, the number of generations required to get the model was good enough but not too large indicating there was no time given for the model to overfit.

Some Tricks

- On careful analysis of the overfit array by the naked eye, we saw that coefficients at index 7, 8 and 9 were larger than others therefore had more say in the errors. Therefore we mutated these indexes less violently and it straightaway improved our model. This change also gave us our first breakthrough model.
- One trick which we also would like to mention is that, often starting off again and again with the initial population seems to be a cumbersome task. So what we did to save time and requests is we dumped the last generation's vectors ,training errors and validation errors in a file in json format. And if we wish to start off with the last generation we load this data to our main.py file for subsequent use.This not only saved our requests(which were quite low) but also saved our time because we can start from our last left generation preventing time to get this population back.