Genetic Algorithm Project Report

TeamXYZ
Ananya Amancherla (2019101041)
Samruddhi Shastri (2019111039)

Summary

The genetic algorithm is a search heuristic inspired by Charles Darwin's theory of natural evolution. The algorithm reflects the process of natural selection where the fittest individuals of a generation are selected for reproduction in order to produce offspring of the next generation.

Step by step review of the algorithm:

Step 1: Generating Initial Population:

To start the algorithm, we require a population of size pop_size. These individuals are generated by mutating a starting vector - either the original vector from the overfit.txt file provided to us or a better vector obtained in one of the previous runs of the program.
We have ensured that the first generation of individuals is diverse by keeping the probability of mutation factor high, that is the probability with which a gene is
mutated in a chromosome is high. This is so that the algorithm does not converge to a local minima in the very beginning itself

Step 2: Get Errors for Current Population:

To define the fitness of any individual and then select which parents must be chosen to produce offspring, we need to obtain the train/validation errors of the individuals of the population using the get errors function. The fitness function is defined as train error+ (factor)* validation error; where we alternated the factor between 1.5 and 1.

Step 3:Check Minimum Error:

We compare the fitness value of each individual with min_err to find the min_err among all individuals of all generations. We also store the corresponding solution vector and the generation number to facilitate writing into the file.

Step 4: Create New Population and Generate Mating Pool:

The new population for this generation is defined by combining the parents of the previous generation along with the population in the current generation. For the first iteration, new_pop is simply the initial population.A mating pool of MATING_POOL_SIZE is selected from the sorted(by fitness function) population.

Step 5: Crossover and Mutation:

2 parents at a time are chosen from the mating pool randomly and either single point, multi point or simulated binary crossover to produce two children. These children are then mutated, and stored as part of the new population.

Diagrams for 3 Consecutive Iterations

Generation 1

FITNESS FUNCTION

9845727881 10.7854 25%

11759151672 20.6125 25%

13664597304 27.5168 25%

14530074692 17.4705 25%

AFTER SELECTION

AFTER CROSSOVER

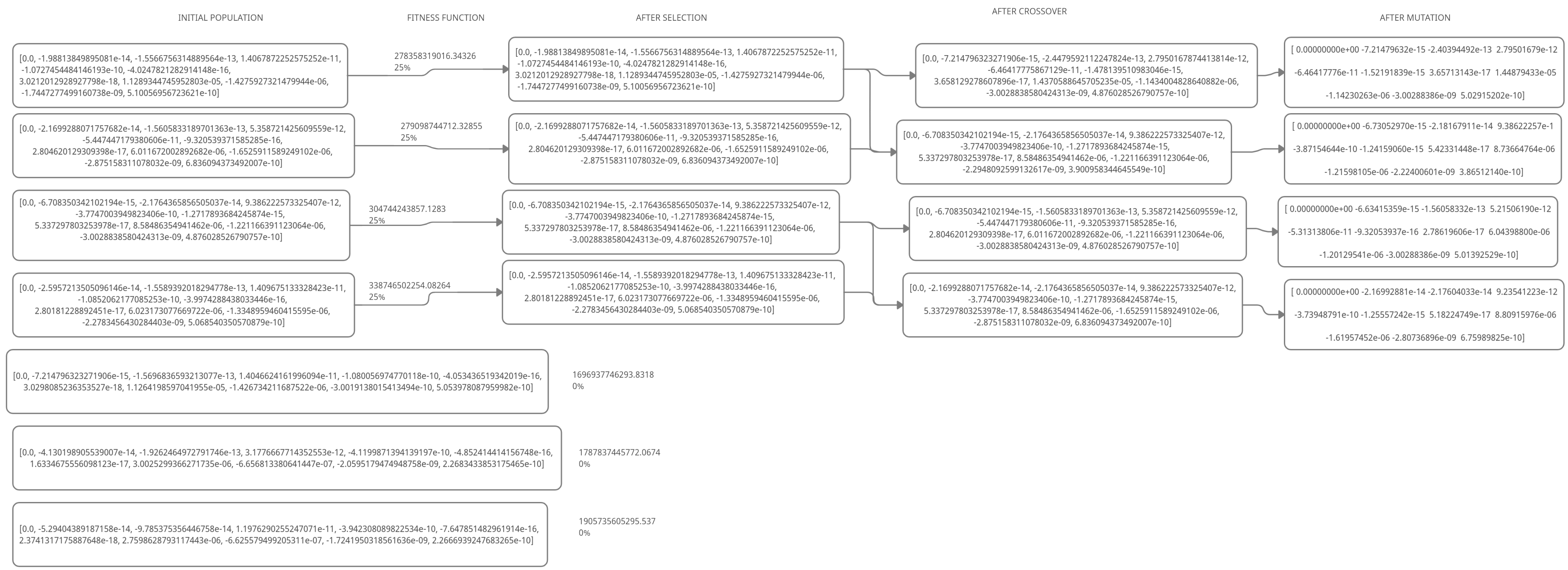
AFTER MUTATION

1437247232 14.220.44 0%

1439339538 84.944.44 0%

1456252120 64.192.8 0%

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 value of 'Train Error * Train Factor + Validation Error'.

We varied the train error from 1.5 for the first k generations and reduced it to 1 for the later ones. It was done because the given vector overfits the training data so, more priority is given to validation errors in the initial generations. Later, equal importance is given to both of them because the population isn't as random in the later iterations and both train and validation errors are equally important for fitness.

Crossover Function

For crossover, we started with single point crossover but soon noticed that it wasn't much effective. Then we replaced it by two point crossover. It increased the variance within the population. For further improvement, we used simulated binary crossover.

In the single/multi point crossover, we determined the crossover point(s) randomly and then appended the relevant portions of the parent vectors to the child vectors.

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

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

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

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

Mutations

The mutateall function takes 2 parameters: temp and mutate_range where temp is the vector to be mutated and the every weight in the vector is multiplied by a factor in the range(1-mutate_range, 1+mutate_range) while accounting for the range of the vectors.

We used the mutation functions in 2 places: once while generating the initial population where we kept mutate_range constant at 0.9 or 0.95.

During the iterations,we used mutateall after the crossover where we selected mutate_range randomly from (0,1) and started with a prob_mut_cross of 0.7; increasing it by 0.01 or 0.02 every 2 iterations.

Hyperparameters

- **Population size** : Initially we started with a population size of 10. As we progressed in the assignment, we were able to better our GA and realised that a population size of 30 suited the best to provide a diverse enough population to choose from. In the end we used a population size of 60.
- **Mating pool size:** For a population size of 30, we used a mating pool size of 15 and then 10 to refine the selected vectors. For a population size of 60 we used a mating pool size of 20.
- **Mutate Range** : We have set this parameter to 0.1. The overfit vector is sensitive and if the mutation is drastic it will lead to a high error. Hence we made sure that the elements of the vector undergoing mutation change by this formula :

  With simulated annealing, this range decreases by 0.01 every 2 iterations and prob_mut_cross increases by 0.01 or 0.02.
- **Mutation Probability** : This parameter is set to 0.7 to start with a large degree of mutations. This will ensure diversity and prevent converging to a local minima. Further the mut_prob increases every 2 iterations.

Heuristics

While constructing the Genetic Algorithm, the heuristics that we applied include :

1. Using a fitness function with weights : We ran the algorithm with a fitness function as follows : err = err[0] + 1.5 * err[1]. This function however did not seem to help our algorithm to converge to the global minima. The train and validation errors we received were still quite high. To try and improve the algorithm, we also tried changing the fitness function midway. For first k iterations we had a fitness function of err = err[0] + 1.5*err[1]. After k iterations, we changed the fitness function to simply err = err[0] + err[1]. We did so considering that with a 'not so random' population in the later iterations of the GA, we can see that both training and validation errors are equally important and one cannot overpower the other in deciding the fitness of an individual.

2. Simulated Annealing : In this method, we reduce the range within which a particular gene/element of the chromosome/vector can be mutated. We start off with a mutate_range of 0.1. After every 2 iterations, this is decreased by 0.01. As a consequence of the decreased mutate_range the vectors may now come very close to each other. To prevent the algorithm

from converging to a local minima, we started off with a prob_mut of 0.7 and increased by 0.01 every 2 iterations. This method helped us in the process of achieving the global minima.

Statistical Information

We initially used a population size of 10 and the vectors converged to a value with a very high fitness function after ~120 generations. On increasing the population size to 30 and creating the new population from both the current generation and previous generation, we reached a local minima in ~100 generations.

Then we introduced simulated annealing and made minor changes to mating pool size ~20 generations.

Finally we increased population size to 60 and got the optimum vector in 10 generations.

Train Error and Validation Error

Final Vector: [0.0, -1.9857615430189556e-14, -1.598849168384364e-13, 1.2341187203924752e-11, -1.0852062177085253e-10, -3.9974288438033446e-16, 2.840417052027975e-17, 5.977920246542961e-06, -1.301258086466706e-06, -2.2689048910190943e-09, 5.051582830433647e-10]

Fitness Function corresponding to vector: 99592217339.66658

We believe this could be the vector on the test server as this is a vector from one of our last populations and has the best fitness function value we ever got. It is one of the vectors we submitted on the day we got the improved result.