Genetic Algorithm Project

Team Number : 29

Team Members :

Manasvi Vaidyula 2019101012

Pooja Desur 2019101112

Summary

A genetic algorithm is a search heuristic (related to making guesses) algorithm that is inspired by Charles Darwin's theory of natural evolution (related to how life evolves). This algorithm reflects the process of natural selection where the fundamental idea is that the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

Algorithm Implementation

1 : Generating Initial Population

- We start the algorithm by generating a population of size `POP` by mutating the `overfit_vector` given to us.

```
def generate_initial(overfit_vector):

    generation = np.zeros(shape = (POP,FEATURE))
    for i in range(POP):
        for feature in range(FEATURE):

            val = vector[feature]
            prob = random.uniform(0,1)
            if(prob <= 0.9):

                delta = random.uniform(MUTATE_RANGE[0],MUTATE_RANGE[1])
                val = val * delta

            generation[i][feature] = val

    return generation
```

- We put the probability of mutating a feature in the vector as 0.9 so as to get a diverse initial polulation.
- Each feature is mutated by factor from the range `MUTATE_RANGE`
- Each vector in the generated population is of size `FEATURE`

We sorted the population by their fitness and saved the data in the following format. The parents index helped to see which parents were selected during the selection process, and which children it created in the crossover stage.

GENERATION FITNESS AND ERRORS

INDEX	POPULATION	TRAINING ERRROR	VALIDATION ERRROR	FITNESS
P0	[0.00000000e+00 -1.50529678e-12 -2.06233352e-13 4.41314958e-11 -1.85774715e-10 -1.94022580e-15 8.70283624e-16 2.40328555e-05 -2.19509741e-06 -1.44710288e-08 1.01610160e-09]	9.79892e+10	2.842e+11	3.8219e+11
P1	[0.00000000e+00 -1.45808017e-12 -2.28980078e-13 4.57212158e-11 -1.63066471e-10 -1.69129453e-15 7.82262882e-16 2.14164119e-05 -2.10187591e-06 -1.69370627e-08 1.01407262e-09]	6.17689e+11	7.87629e+11	1.40532e+12
P2	[0.00000000e+00 -1.40284660e-12 -2.28980078e-13 4.53077768e-11 -1.63304827e-10 -1.93322708e-15 8.50068889e-16 2.29423303e-05 -1.92451222e-06 -1.59792834e-08 9.04815953e-10]	8.8526e+11	8.10493e+11	1.69575e+12
P3	[0.00000000e+00 -1.57146213e-12 -2.20394289e-13 4.60447548e-11 -1.75214813e-10 -1.66179795e-15 7.77495825e-16 2.29423303e-05 -2.04721003e-06 -1.69739835e-08 9.69223672e-10]	9.12283e+11	1.08795e+12	2.00024e+12
P4	[0.00000000e+00 -1.32467529e-12 -2.51579697e-13 4.75971323e-11 -1.65152235e-10 -1.74281832e-15 8.02700894e-16 2.35742760e-05 -1.92428125e-06 -1.64823010e-08 8.98728449e-10]	1.12581e+12	1.19211e+12	2.31792e+12
P5	[0.00000000e+00 -1.53825161e-12 -2.36821564e-13 4.62010753e-11 -1.60658819e-10 -1.66905039e-15 7.92570567e-16 2.29447360e-05 -2.12010252e-06 -1.61953068e-08 1.06053226e-09]	4.75146e+11	1.91208e+12	2.38723e+12
P6	[0.00000000e+00 -1.43884852e-12 -2.38572346e-13 4.68915622e-11 -1.70577522e-10 -1.83369073e-15 7.76263811e-16 2.47299541e-05 -2.11863426e-06 -1.69847725e-08 9.77384316e-10]	1.433e+12	1.72116e+12	3.15416e+12
P7	[0.00000000e+00 -1.48758103e-12 -2.37345652e-13 4.49944308e-11 -1.88747376e-10 -1.85920683e-15 8.64723787e-16 2.29423303e-05 -2.13309635e-06 -1.44529240e-08 1.04380696e-09]	7.56107e+11	2.74699e+12	3.5031e+12
P8	[0.00000000e+00 -1.60084491e-12 -2.28980078e-13 4.53410309e-11 -1.76718299e-10 -1.71731472e-15 8.21480913e-16 2.19393818e-05 -2.12455350e-06 -1.59792834e-08 1.07913009e-09]	8.56881e+11	2.89448e+12	3.75136e+12
P9	[0.00000000e+00 -1.48768149e-12 -2.22424632e-13 4.21534642e-11 -1.67462734e-10 -1.73068200e-15 8.89314489e-16 2.37961542e-05 -2.21140691e-06 -1.68573301e-08 1.01585036e-09]	1.63929e+12	2.22405e+12	3.86334e+12
P10	[0.00000000e+00 -1.34472312e-12 -2.22982066e-13 4.21847409e-11 -1.67617298e-10 -1.73012244e-15 9.38010718e-16 2.24938141e-05 -1.89950453e-06 -1.59792834e-08 9.82527580e-10]	1.00186e+12	3.25175e+12	4.25361e+12
P11	[0.00000000e+00 -1.45799022e-12 -2.26133948e-13 4.70463235e-11 -1.73243436e-10 -1.77042462e-15 8.52944060e-16 2.16157072e-05 -2.24271012e-06 -1.50311515e-08 9.97607886e-10]	2.51456e+12	3.59829e+12	6.11285e+12
P12	[0.00000000e+00 -1.36477829e-12 -2.50816895e-13 4.53225682e-11 -1.57964542e-10 -1.99028745e-15 8.52944060e-16 2.10385122e-05 -1.96027067e-06 -1.62028982e-08 1.04417200e-09]	1.59779e+12	4.78678e+12	6.38457e+12
P13	[0.00000000e+00 -1.51199066e-12 -2.47974644e-13 4.82290691e-11 -1.87229167e-10 -1.65553069e-15 7.84007498e-16 2.22659363e-05 -2.19619911e-06 -1.50951439e-08 9.64505101e-10]	2.93955e+12	4.26223e+12	7.20178e+12
P14	[0.00000000e+00 -1.52380687e-12 -2.07902211e-13 4.47253523e-11 -1.74746657e-10 -1.82067602e-15 8.77946901e-16 2.46163663e-05 -2.06366171e-06 -1.66894814e-08 1.07118832e-09]	2.02813e+12	5.78704e+12	7.81517e+12
P15	[0.00000000e+00 -1.45799022e-12 -2.20963595e-13 4.68942503e-11 -1.59499181e-10 -1.82169679e-15 8.82338847e-16 2.24575335e-05 -2.03494183e-06 -1.59792834e-08 1.07480859e-09]	2.80303e+12	7.49407e+12	1.02971e+13
P16	[0.00000000e+00 -1.51805889e-12 -2.30743390e-13 4.73534345e-11 -1.65352309e-10 -1.97065909e-15 8.71324921e-16 2.38697916e-05 -1.97038800e-06 -1.72841813e-08 1.07314302e-09]	3.38345e+12	8.93563e+12	1.23191e+13
P17	[0.00000000e+00 -1.55582946e-12 -2.28980078e-13 4.62010753e-11 -1.86673666e-10 -1.94300143e-15 8.32423885e-16 2.19265375e-05 -2.05849342e-06 -1.67102857e-08 9.06680614e-10]	6.80597e+12	1.02838e+13	1.70898e+13
P18	[0.00000000e+00 -1.36066428e-12 -2.41111914e-13 4.47900864e-11 -1.73621306e-10 -2.01017533e-15 9.13340019e-16 2.44432095e-05 -2.01833733e-06 -1.46060664e-08 1.07623015e-09]	7.33488e+12	1.783e+13	2.51648e+13
P19	[0.00000000e+00 -1.47054669e-12 -2.25911290e-13 4.72263836e-11 -1.87667677e-10 -1.66286120e-15 9.30338715e-16 2.35465305e-05 -2.22369617e-06 -1.68086235e-08 9.04480525e-10]	1.40217e+13	2.33918e+13	3.74135e+13

2 : Retrieving Errors And Determining Fitness

- This is the step where we determine the fitness of every vector in the population.

```
def get_fitness(generation,type):

    fitness = call_server(generation)
    fitness, generation = fitness_function(fitness, generation,type)

    return fitness, generation
```

- First we retrieve the training and validation errors of each vector by making calls to the server.

```
def call_server(generation):

    fitness = np.zeros(shape = (POP,3))
    for i in range(POP):
        error = get_errors(SECRET_KEY,generation[i].tolist())
        fitness[i][0] = error[0]
        fitness[i][1] = error[1]

    fitness = np.column_stack((generation,fitness))
    return fitness
```

- Next we determine the fitness of the vector by taking some linear combination of both the dervied errors.

  Fitness = `TRAINING_FACTOR` * training_error + `VALIDATION_FACTOR` * validation_error

- These vectors are then sorted in increasing order of their fitness value.

```
def fitness_function(fitness, generation,type):

    for i in range(POP):
        fitness[i][FEATURE+2] = (TRAINING_FACTOR*fitness[i][FEATURE] + VALIDATION_FACTOR*fitness[i][FEATURE+1])

    sorted_idx = np.argsort(fitness[:,-1])
    fitness = fitness[sorted_idx]
    generation = generation[sorted_idx]
```

3 : Selection

- A mating pool is generated by selecting the top `POOL_SIZE` vectors of the current generation

```
def selection(generation):

    pool = np.zeros(shape = (POOL_SIZE,FEATURE))
    pool = generation[:POOL_SIZE]

    return pool
```

SELECTED MATING POOL

INDEX	PARENT
P0	[0.00000000e+00 -1.50529678e-12 -2.06233352e-13 4.41314958e-11 -1.85774715e-10 -1.94022580e-15 8.70283624e-16 2.40328555e-05 -2.19509741e-06 -1.44710288e-08 1.01610160e-09]
P1	[0.00000000e+00 -1.45808017e-12 -2.28980078e-13 4.57212158e-11 -1.63066471e-10 -1.69129453e-15 7.82262882e-16 2.14164119e-05 -2.10187591e-06 -1.69370627e-08 1.01407262e-09]
P2	[0.00000000e+00 -1.40284660e-12 -2.28980078e-13 4.53077768e-11 -1.63304827e-10 -1.93322708e-15 8.50068889e-16 2.29423303e-05 -1.92451222e-06 -1.59792834e-08 9.04815953e-10]
P3	[0.00000000e+00 -1.57146213e-12 -2.20394289e-13 4.60447548e-11 -1.75214813e-10 -1.66179795e-15 7.77495825e-16 2.29423303e-05 -2.04721003e-06 -1.69739835e-08 9.69223672e-10]
P4	[0.00000000e+00 -1.32467529e-12 -2.51579697e-13 4.75971323e-11 -1.65152235e-10 -1.74281832e-15 8.02700894e-16 2.35742760e-05 -1.92428125e-06 -1.64823010e-08 8.98728449e-10]

4 : Crossover

- In this step we generate a crossOver population of size `POP` by crossing over the parents from the pool array.

```
def crossover(pool):

    crossOver_generation = np.zeros(shape= (POP,FEATURE))
    i = 0
    while i < POP:

        p1 = random.choice(list(enumerate(pool)))
        p2 = random.choice(list(enumerate(pool)))

        u = random.uniform(0,1)

        if (u < 0.5):
            b = (2 * u)**((NC + 1)**-1)
        else:
            b = ((2*(1-u))**-1)**((NC + 1)**-1)

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

        crossOver_generation[i]= child1
        crossOver_generation[i+1] = child2

        i += 2

    return crossOver_generation
```

- Parents are selected from the pool size uniformly.
- We apply **Simulated Binary Crossover** on the 2 selected parents to generate 2 children.

5 : Mutation

- We now apply mutation on the vectors created in the crossOver generation.
- Probability of mutating a feature is `MUTATE_PROB`. The feature is multiplied by a factor of `MUTATE_RANGE`

```
def mutation(crossOver_generation):
    i = 0
    for child in crossOver_generation:
        for feature_index in range(FEATURE):

            prob = random.uniform(0, 1)
            if(prob <= MUTATE_PROB):

                delta = random.uniform(MUTATE_RANGE[0],MUTATE_RANGE[1])
                new_feature = child[feature_index]* delta

                child[feature_index] = new_feature

    return crossOver_generation
```

- Next We repeat `step 2` again this time with generated children population and sort based on fitness value.

We saved the data in the following format to see which parents created which child, and the child after mutation.

6 : New Generation

- From the `POP` size parent and children that we currently have we create our next generation.
- We select the top `PARENT` vectors from the parent generation and top `CHILD` vectors from the children generation. (`CHILD = POP - PARENT`)
- We do this so that best features from both parents and children are carried forward to the next generation.

```
def create_newGeneration(parents_fitness,children_fitness,iter):

    parents_fitness = parents_fitness[:PARENT]
    children_fitness = children_fitness[:CHILD]
    new_fitness = np.concatenate((parents_fitness,children_fitness))

    sorted_idx = np.argsort(new_fitness[:,-1])
    new_fitness = new_fitness[sorted_idx]
    new_generation = new_fitness[:,:FEATURE]
```

We repeat steps 2 to 6 for `ITERATION` number of iterations.

We used the following format to view the children created along with their respective errors and fitness.

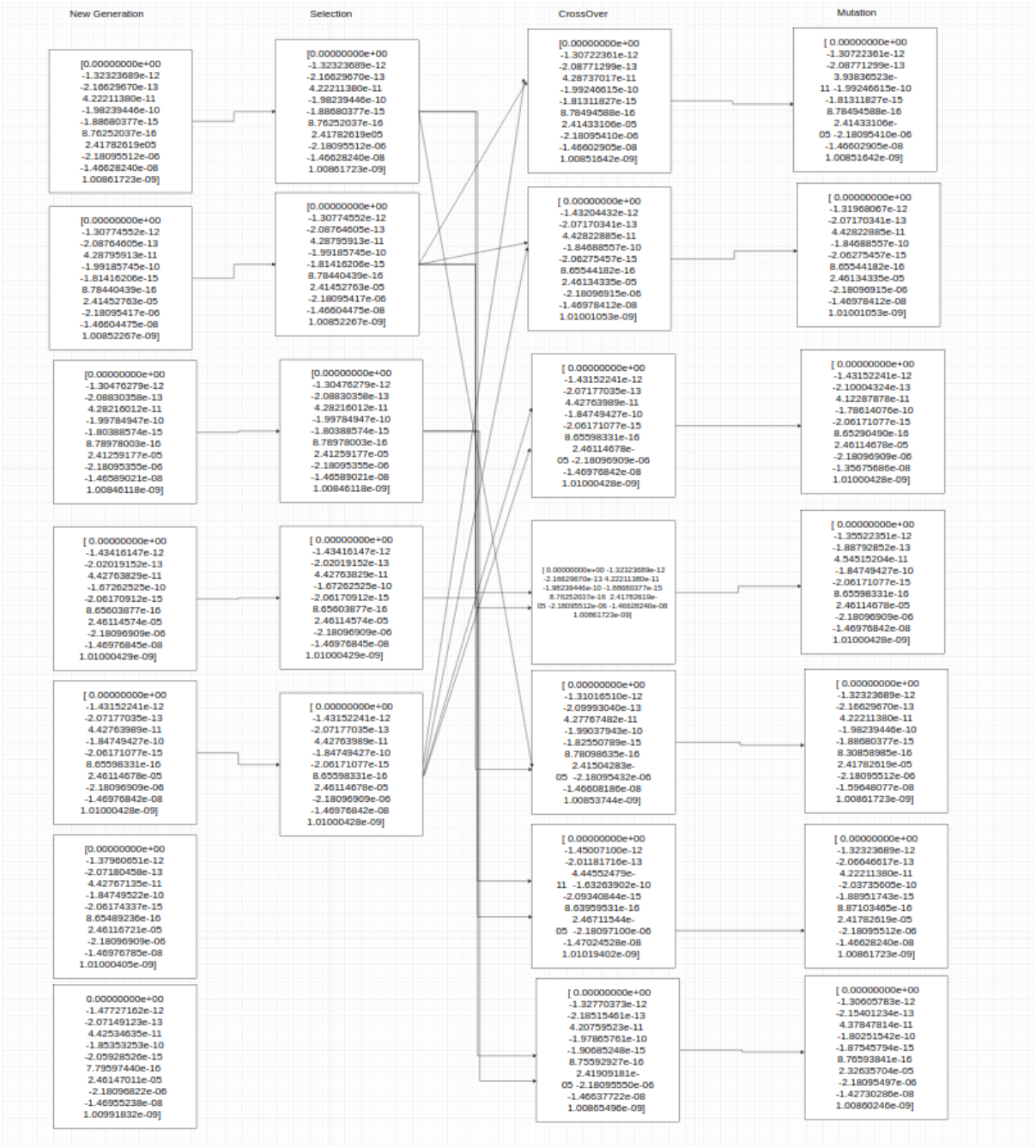
CHILD GENERATION FITNESS AND ERRORS

INDEX	POPULATION	TRAINING ERRROR	VALIDATION ERRROR	FITNESS
C0	[0.00000000e+00 -1.64754168e-12 -2.00589465e-13 4.43714334e-11 -1.82347343e-10 -2.02944494e-15 8.92573276e-16 2.36379537e-05 -2.18102742e-06 -1.48432291e-08 1.01579536e-09]	1.48954e+11	2.35203e+11	3.84157e+11
C1	[0.00000000e+00 -1.40473390e-12 -1.88941359e-13 4.40242605e-11 -1.86366582e-10 -1.95583128e-15 8.75484244e-16 2.40939778e-05 -2.20338628e-06 -1.43307418e-08 1.01872984e-09]	9.49579e+10	3.19247e+11	4.14205e+11
C2	[0.00000000e+00 -1.50653308e-12 -2.00160549e-13 4.48856452e-11 -1.81287104e-10 -2.07312293e-15 8.55577099e-16 2.26235769e-05 -2.13616572e-06 -1.49086972e-08 1.00248584e-09]	2.12682e+11	2.48433e+11	4.61114e+11
C3	[0.00000000e+00 -1.53676207e-12 -1.98333761e-13 4.35277622e-11 -1.89367268e-10 -1.97461531e-15 8.82056920e-16 2.41127427e-05 -2.24227513e-06 -1.41206537e-08 1.03654867e-09]	1.03244e+11	4.54204e+11	5.57448e+11
C4	[0.00000000e+00 -1.45523017e-12 -2.30353076e-13 4.96517056e-11 -1.67053586e-10 -1.67626898e-15 7.76949928e-16 2.12584827e-05 -2.09624903e-06 -1.63115984e-08 1.01395015e-09]	2.44095e+11	3.64325e+11	6.0842e+11
C5	[0.00000000e+00 -1.32073164e-12 -2.29974536e-13 4.52224155e-11 -1.68573319e-10 -2.15583389e-15 8.58474742e-16 2.29423303e-05 -1.91030061e-06 -1.48844065e-08 8.97355860e-10]	4.11885e+11	2.43984e+11	6.55869e+11
C6	[0.00000000e+00 -1.60051247e-12 -2.26611746e-13 4.68847843e-11 -1.70578416e-10 -1.53955229e-15 7.36756706e-16 2.24635277e-05 -1.90795785e-06 -1.80729229e-08 9.48641597e-10]	4.60497e+11	4.17002e+11	8.77499e+11
C7	[0.00000000e+00 -1.46520661e-12 -2.25546898e-13 4.54812782e-11 -1.66493843e-10 -1.72886592e-15 7.95547918e-16 2.18113136e-05 -2.11594590e-06 -1.65648624e-08 1.05892010e-09]	1.04561e+11	8.26166e+11	9.30727e+11
C8	[0.00000000e+00 -1.65193659e-12 -1.96139591e-13 4.17834383e-11 -1.47923634e-10 -1.61642308e-15 7.52563520e-16 1.82807259e-05 -2.35994640e-06 -1.75978968e-08 1.18168425e-09]	2.83097e+11	8.08409e+11	1.09151e+12
C9	[0.00000000e+00 -1.36397993e-12 -2.41711981e-13 4.68429830e-11 -1.70596828e-10 -1.78577572e-15 8.17407419e-16 2.36740664e-05 -1.98321295e-06 -1.60446326e-08 9.24269755e-10]	7.48744e+11	7.21404e+11	1.47015e+12
C10	[0.00000000e+00 -1.13081887e-12 -2.84420183e-13 5.37678891e-11 -1.80291334e-10 -1.81768977e-15 8.32400256e-16 2.67099619e-05 -1.66621076e-06 -1.58214669e-08 7.54947373e-10]	1.03182e+12	7.77849e+11	1.80967e+12
C11	[0.00000000e+00 -1.37853172e-12 -2.28980078e-13 4.51257731e-11 -1.63409756e-10 -2.03973041e-15 8.79918389e-16 2.36140688e-05 -1.84643333e-06 -1.55576506e-08 8.56719078e-10]	1.04814e+12	8.81599e+11	1.92974e+12
C12	[0.00000000e+00 -1.59099220e-12 -2.19399831e-13 4.44866338e-11 -1.73635419e-10 -1.63035938e-15 7.69089972e-16 2.29423303e-05 -2.06142164e-06 -1.70891957e-08 9.76683765e-10]	9.18368e+11	1.12697e+12	2.04534e+12
C13	[0.00000000e+00 -1.42804313e-12 -2.21187989e-13 4.61519902e-11 -1.74622946e-10 -1.64619248e-15 7.72295205e-16 2.28812080e-05 -2.03892116e-06 -1.71142704e-08 9.66596233e-10]	1.00053e+12	1.2131e+12	2.21363e+12
C14	[0.00000000e+00 -1.35343102e-12 -2.43512131e-13 4.75971323e-11 -1.65152235e-10 -1.91210587e-15 8.31703645e-16 2.35742760e-05 -1.92428125e-06 -1.64823010e-08 8.98728449e-10]	1.338e+12	1.39594e+12	2.73393e+12
C15	[0.00000000e+00 -1.48239505e-12 -2.12030366e-13 4.59032196e-11 -1.50164797e-10 -1.58479120e-15 7.52413381e-16 2.07446734e-05 -2.35134622e-06 -1.67952767e-08 1.09959919e-09]	1.53217e+12	2.50438e+12	4.03655e+12
C16	[0.00000000e+00 -1.61264874e-12 -2.05761966e-13 3.98654518e-11 -1.90411112e-10 -2.09164602e-15 8.36215005e-16 2.35978912e-05 -2.07473270e-06 -1.33720893e-08 1.00564907e-09]	1.11875e+12	3.81293e+12	4.93169e+12
C17	[0.00000000e+00 -1.32467529e-12 -2.51579697e-13 4.75971323e-11 -1.65152235e-10 -1.74281832e-15 8.02700894e-16 2.35742760e-05 -2.02923078e-06 -1.54858681e-08 8.98728449e-10]	2.48381e+12	3.22644e+12	5.71025e+12
C18	[0.00000000e+00 -1.29321000e-12 -2.59479288e-13 4.82008659e-11 -1.61559682e-10 -1.70842881e-15 7.90927598e-16 2.15060224e-05 -1.87710354e-06 -1.68326760e-08 8.78281379e-10]	2.52083e+12	3.28071e+12	5.80153e+12
C19	[0.00000000e+00 -1.50814679e-12 -2.04860354e-13 4.40355399e-11 -1.87145390e-10 -1.95525136e-15 8.75596578e-16 2.41907847e-05 -2.20072428e-06 -1.43221783e-08 1.10858254e-09]	3.48381e+12	9.1607e+12	1.26445e+13

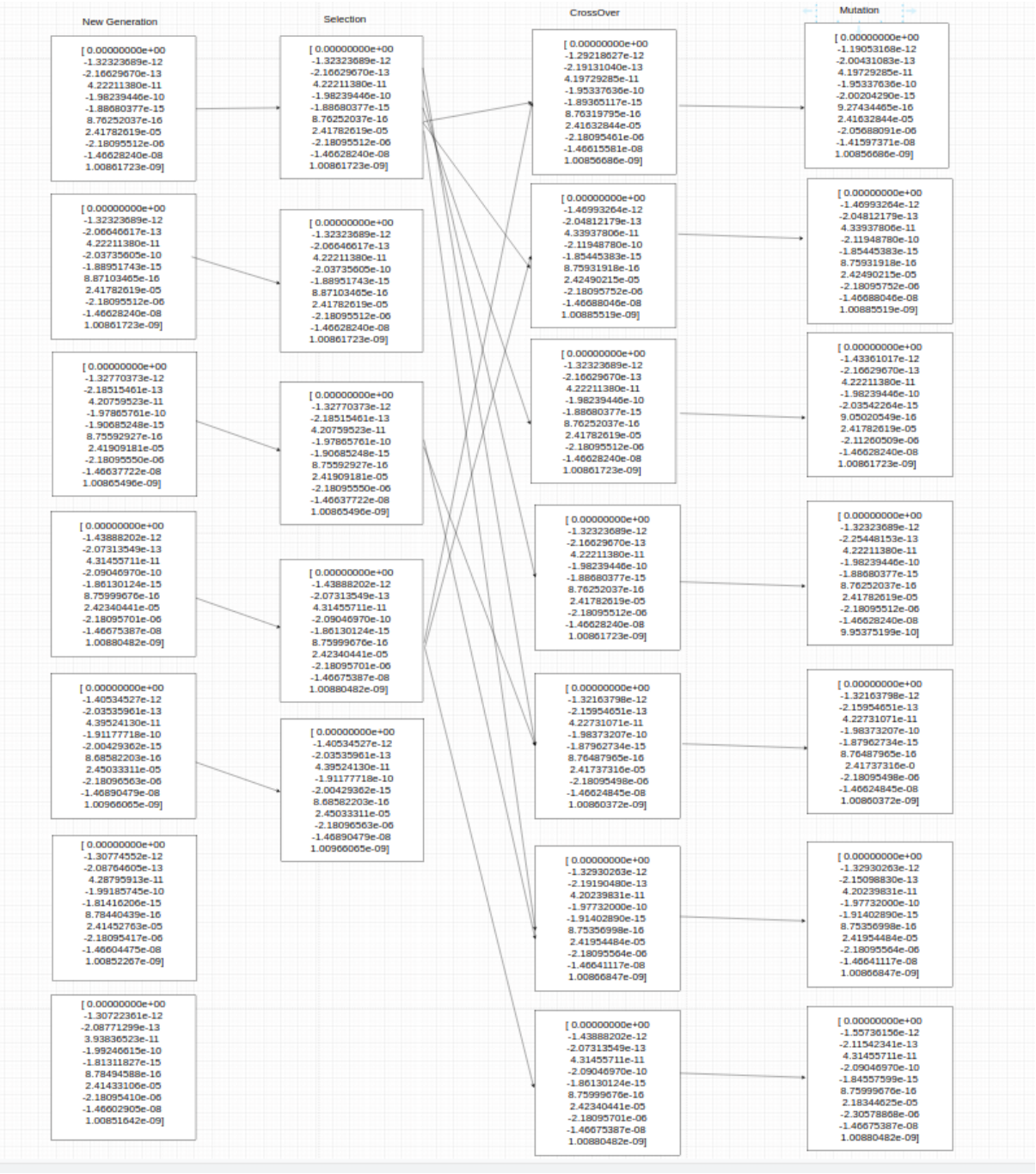
Successive Iterations

The following represents three consecutive iterations of the genetic algorithm we ran among the four stages - the initial generation, selection, crossover and mutation.

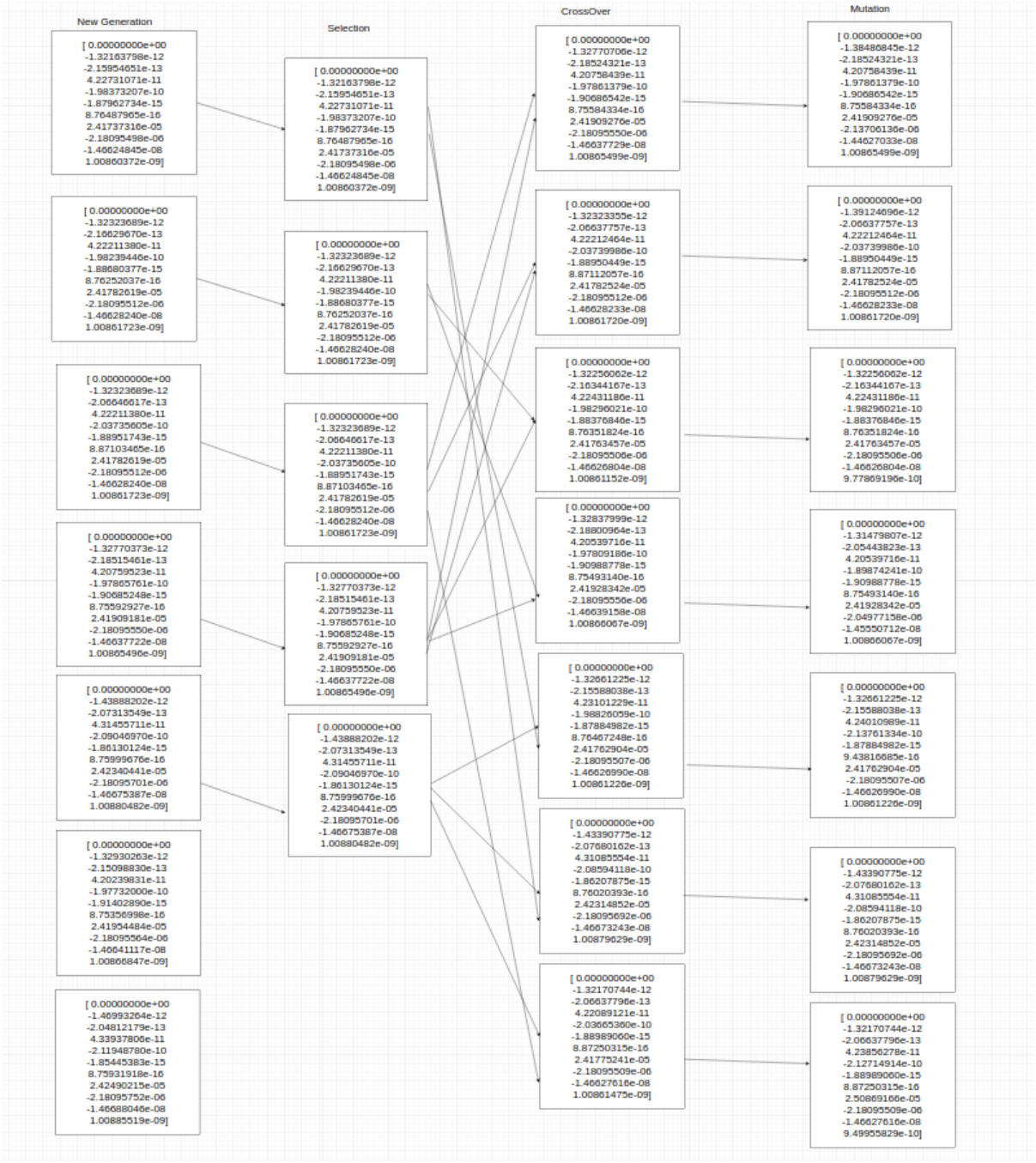
Iteration 9



Iteration 10



Iteration 11



Fitness Function

In order to determine which vectors would perform well on an unseen dataset and not overfit to the training dataset or perform too poorly on the validation dataset, the fitness function is constructed.

On calling the server for a particular vector, its training and validation errors are returned. The relative importance between these two errors, and thus deciding what makes a vector "better" is captured by the fitness function which returns the "fitness" for the vector.

The higher the fitness of a vector, the lesser was the likelihood that it would perform well on an unseen dataset. The fitness helps us choose which parents to select for the mating pool and which vectors created in one iteration of the genetic algorithm will be selected as the next iteration's parent generation.

$$fitness(vector) = training\_factor * training\_error(vector) + validation\_error(vector)$$

For the first 40 iterations the $training\_factor = 0.8$. with population size 20.

This as done as we wanted to reduce validation error by a drastic amount first, and so gave less importance to training error and allowed it to influence the fitness to a lesser degree.

Once we were satisfied that the vectors we were generating had desirable errors (not too low that it was overfitting to both the training and validation datasets, but not so high that it would perform poorly on unseen datasets), the fitness function was altered with $training\_factor = 1$

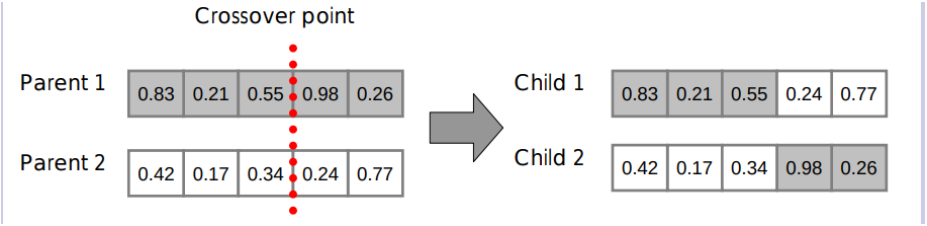
The fitness being calculated as a sum of training and validation errors ensured that training errors did not climb too high as validation errors dropped with the previous fitness function. By giving equal importance to both training and validation errors, vectors with both low training and validation errors were selected while validation errors were improved remarkably from the original overfit vector.

For the next 150 iterations, the training_factor = - 1. Thus the fitness function became the absolute difference of training_error and validation_error. This was done to bring the errors close together while selecting the vectors for the mating pool. By generating vectors that had validation and training errors with miniscule difference we deduced that that vector will generalize well for an unseen dataset.

Crossover

Single Point Crossover

- At the start of the assignment we implemented single point crossover.
- For creating a child the first parent was copied till a certain index point and the remaining features were taken from the second parent.



- For getting the index point we tried 2 ways -
  - split features almsot evenly, i.e, take first 5 features from the first parent and the last 6 features from the second parent.
  - uniformly pick a random point from the range (3, 8) as the index point.
- However after reading a few more articles on the different crossover algorithms we came across Simulated Binary Crossover and tried implementing that instead.

Simulated Binary Crossover

- This algorithm gave us more control over how much variation we wanted.(i.e how different do we want the children to be from the parent)

The algorithm is as follows -

- Select 2 parents $parent_1$ and $parent_2$
- Generate a random number **u** from (0, 1]
- calculate β using the formula-

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

- n_c is defined as the distribution index usually in the range 2 to 5.
- 2 Children are calculated using the formula-

$$child_1 = 0.5 * [(1+\beta)] * parent_1 + (1-\beta) * parent_2]$$

$$child_2 = 0.5 * [(1-\beta)] * parent_1 + (1+\beta) * parent_2]$$

- Having a value n_c = 2 generates children far from the selected parents and n_c = 5 results in lesser variation from the parents. Using this knowledge we manipulated the way we created our child vectors.
- When the vectors were too similar to each other we made n_c = 2 so as to generate vectors with more variation to diversify the population

- When we observed the errors to be steadily descreasing we made n_c = 5 or 4 . This is because we wanted the children to be as close to the parents which were giving good errors.
- This process of manually changing the n_c value significantly helped us from converging which was happening in the single point crossover algorithm.

Hyperparameters

`POP` = 20

`MUTATE_PROB` = 0.3

`MUTATE_RANGE` = [0.9,1.1]

`POOL_SIZE` = 5

`PARENT` = 5

`ITERATIONS` = 120

`NC` = 3

`TRAINING_FACTOR` = 0.7

`VALIDATION_FACTOR` = 1

- `POP` - Represents the population size. This value ranged as the days progressed. We initially maintained a population of size 10 so as to save server calls. However, this changed to 20 so as to have more diversity in our population.
- `POOL_SIZE` - Represents the top parents we pick in the selection process. Kept this value as 5
- `ITERATIONS` - Represents the number of generations we created. Ran this algorithm for 120 generations.

Statistical Information

Number of Iterations to Converge

Approach 1

When we used random splicing as our crossover with a population size of 10, and a fitness function of only sum, the genetic algorithm started to converge within 50 iterations. We knew we were finding a local minimum and had to change tactics.

Approach 2

By increasing population size and using simulated binary crossover, the genetic algorithm did not converge till the 202nd iteration. Vectors at this point were very similar with miniscule differences in one or more features (as we had intended), but having a mutation probability of 0.3 ensured we weren't getting stuck in a local minumum.

Mutations

Mutation probability is the probability with which each feature in a vector would get altered. Mutation range is a range of the mutation factor by which it would get altered.

$$new\_feature = original\_feature * mutation\_factor$$

As we used the simulated binary crossover approach, we had some level of control of what kind of children were being created i.e we could choose how similar the children generated were to the parent vectors.

Hence, we avoided mutating by too much to prevent the the children created from the crossover from becoming too random.

We mutated each feature in a generated child vector with a probability of 0.3 and a range of 0.9 to 1.1 of the original feature in the vector for the first 40 iterations, and then we started to decrease the mutation range to 0.95 to 1.05. As the vectors started reaching the desired errors, we wanted to prevent too much variation which would end up creating poorly performing children.

Heuristics

General Idea - talk about mid eval?

- Validation and training errors had to be close to each and other and their absolute difference should as low as possible. This would result in a vector that could generalize on an unseen dataset well.
- Our original idea was to drop the validation and training errors as low as possible and try to bring them closer together. With this approach, we reached validation errors of around 5e10 and training errors of around 4e10, and as the original overfit vector had validation and training errors of 3e11 and 1e10 respectively, we felt this was suitable. However, we realized these low errors might be a result of overfitting on both the validation and training datasets. Thus we scrapped this idea and instead tried to attain errors that were significantly lower than the overfit vector's validation errors, but were not so low that it would result in overfit vectors again. We began to target errors of around the range 1e11.

Initial Population

- In our first attempt, we created the initial population by allotting random float values from [-10,10] to each feature of the vector. We quickly realized the major drawbacks of this when produced vectors which had errors in the orders of 10e30.
- We tried creating the initial population with all zeroes, and mutating the vector's features with a probability of 0.7, in a range of 0.9 to 1.1 of the given overfit vector. As the overfit vector had very miniscule values as its features (in the range of 10e-13), we tried to create vectors completely different as we knew this was overfit and did not perform well. This method did not have any advantages though, and we found that creating an initial population by mutating the overfit vector itself with the same range and mutation probability produced better vectors with less runs of the genetic algorithm.

Population size

- When the errors began to converge, we started looking through research papers and resources online. We found that when we changed our initial population size from 10 to 20, we ended up converging much later in the genetic algorithm. This was because more vectors being created led to more diversity.

Fitness Function

- In our first approach, we started off by allowing the fitness function to be the sum of training and validation errors. This gave equal importance and weightage to both errors. The higher the fitness, the more poorly the vector was performing (higher errors).
- In our subsequent approaches, in order for validation to drop by a substantail amount, we then multiplied training error with a training factor of 0.7. This led the validation error to drop while training errors increased. We experimented with different values of training factors and tried with values of 0.5, 0.6 and 0.8, and found that 0.7 gave us lower validation error without letting training errors climb too much.
- To prevent training errors from climbing too high, after 40 iterations, we changed the fitness function to a sum.
- Once we were satisfied that the errors were decent, we changed the fitness function to an absolute difference of training and validation errors to bring them closer together.
- We attempted to bring them closer by using a fitness function of

```
training_error + validation_error + abs(training_error - validation_error)
```

but this led to training errors becoming higher than validation errors which is not desirable as this represents a poor model.

- We achieved bringing them closer together by equating fitness as `abs(training_error - validation_error)`

Selection

- Originally, we attempted to use a weighted probability to select the vectors that would make up the mating pool. The higher the fitness was, the lower the chance the vector would get selected.
- This method did not seem improve errors much or create desirable children to such a drastic extent, so we ended up picking the top POOL_SIZE vectors of the population and used those in simulated binary crossover to generate the children.

Crossover

- We tried various crossover techniques before settling on simulated binary crossover.
- In the first few days, we did a simple crossover from indices 0 to 5 of the first vector and 6 to 11 of the second vector. Two parents would create two children. We did not allow two parents to both be the same vector.
- We then felt this would not provide enough variation in the vectors and so we attempted a random splice. Two unique parents were selected and a random index from 2 to 5 was chosen which marked the middle point the vectors would switch features for.
- In the end, simulated binary crossover ended up dropping the errors drastically, so we adopted that method.

- During the first 40 iterations, we kept `nc` as 2. As we had not created good enough vectors yet, we wanted a large variety with a lot more diversity so chose to create them far from their parents.

Mutation

- As the vectors started performing better and returning better errors, we reduced the mutation range to ensure only well performing children were generated. We did not want to generate children too far from the ones we knew performed well.
- We experimented if increasing the mutation range or decreasing the mutation range as we got further in the genetic algorithm would help. Increasing the mutation range would generate vectors much more different than the best child vector created so far, and could possibly prevent convergence as the algorithm would incorporate these much different vectors as well. However, decreasing the mutation range meant we were satisfied with the best vectors the algorithm had discovered so far and thought the best vector had to be somewhere close to what had already been discovered. With around 20 iterations ran for both conditions, we found that increasing the range produced better errors than decreasing. However, we did not implement this into our final algorithm once we adopted simulated binary crossover.
- When we were using random splicing to generate our children, we increased the mutation probability from 0.7 at the start to 0.85 as the iterations increased in order to prevent convergence.

New Generation

- Our first approach was to take the top `POP`/2 best parents of the previous generation and the top `POP` /2 best children generated to create the new generation. This would not take advantage of the fact that a larger number of better children were created.
- We decided to take the top `PARENT` vectors of all the parents of the previous generation and top `CHILDREN` generated children.

Tricks

- During the first few weeks, we plotted the fitness, training and validation errors, to observe the overall trend and see if a convergence would occur, so we knew if we had to adopt other methods.
- While running our code, we saved the data in two separate files - a trace file which consisted of all the data for each vector - the parents it generated from, how it mutated, each iterations generation, mating pool, children generated, etc.
- The other file was the best vector file which had the best vector for that current fitness function in each iteration. This was helpful as we could gauge whether our algorithm was going on the right track and if there was a convergence or even if the errors started to increase or act erratically.
- Due to a daily server call limit, we saved the best vectors we obtained each iteration in a json file and the next run of the algorithm would pick up the vector from this json file . ie we did not create a new population every day.

- Our biggest idea that helped us was we ran only 8 iterations at a time and observed how the data changed and errors returned and made required changes we felt necessary (changing `mutation range`, `mutation probability`, `nc` and `fitness function`). This way, we had more than one shot a day, and did not use up all our server calls at one shot.