Machine, Data and Learning

TEAM 21
Sai Shashank Kalakonda (2018111016)
Sachin Kumar (2018101108)

Assignment 3 Report

Overview

Generalize an overfit model using genetic algorithms to reduce overfitting. Fitness function should be formulated by querying the training and validation sets.

Genetic Algorithm

Genetic Algorithms (GA) are stochastic algorithms whose search methods model some natural phenomena based on genetic inheritance and natural selection. It is a multi-directional search by maintaining a population of potential solutions and assures information formation and exchange between these directions.

The potential solutions to a problem evolve to a **better-fit** group of solutions . At each generation the better solutions reproduce, while the relatively bad solutions eventually die off.

Genetic algorithm is applied for given problem by us in the following way:

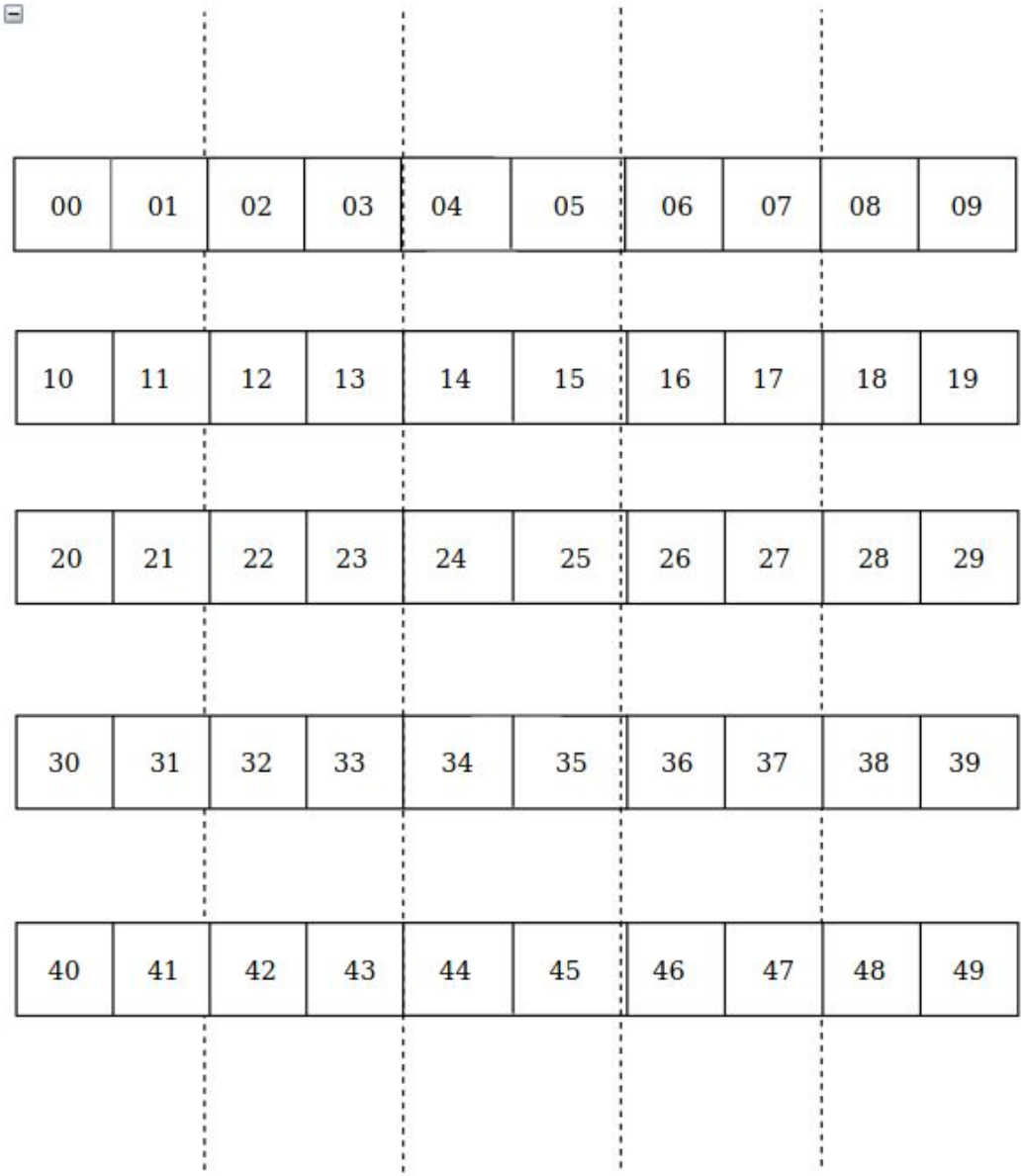
1. Initial population with 10 vectors is created(reference from given overfit vector).
2. Using get_errors functions,to obtain train error and validation error for each vector. Then fitness function is calculated using those errors.
3. Now the top half of the population with better fitness is selected for crossover.Here we used **Diagonal multi-parent crossover function**.new generation consists of Five best parents of previous generation and their crossovers.
4. In each generation, it is subjected to a uniformly random mutation.

5. These generations continue till they converge.

Steps performed in each iteration of GA:

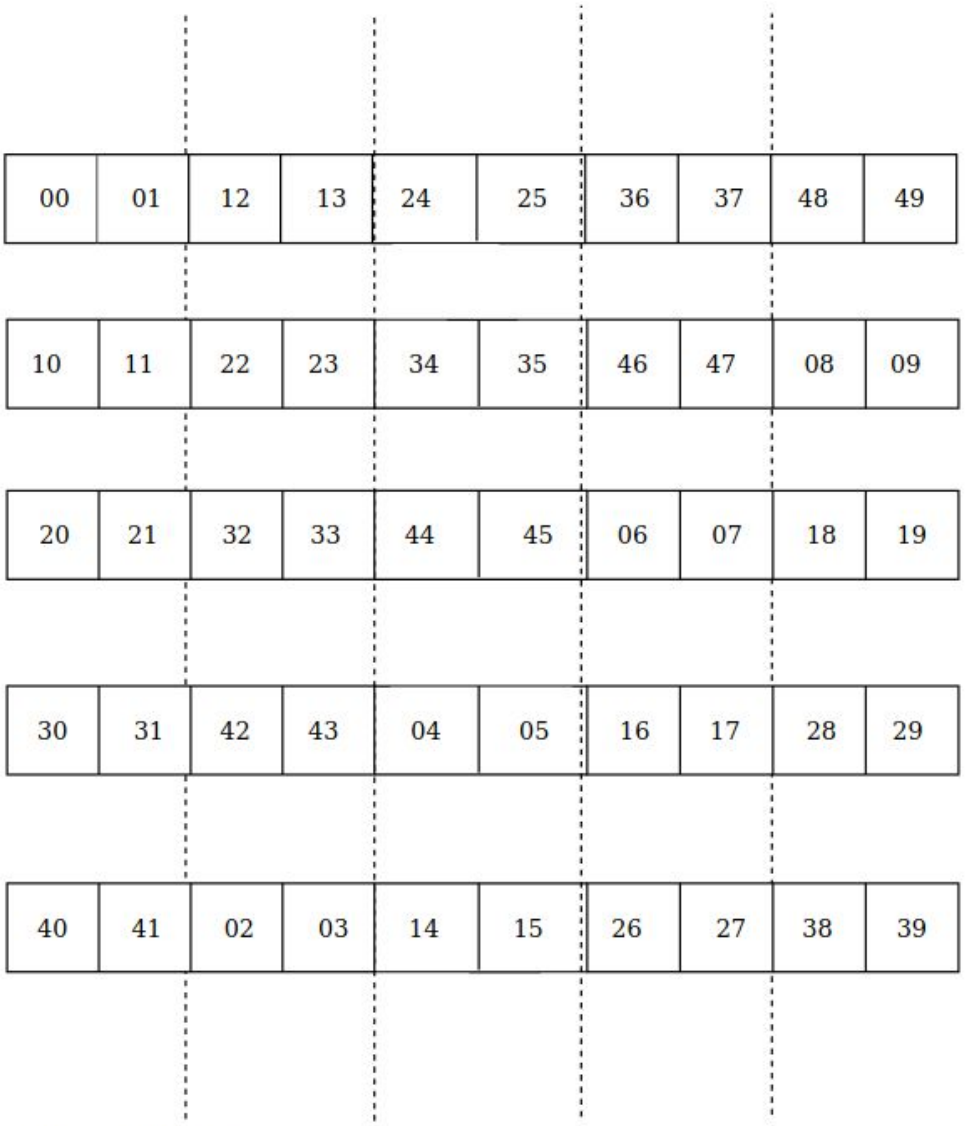
00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Initial Population(sorted by fitness)



Selection

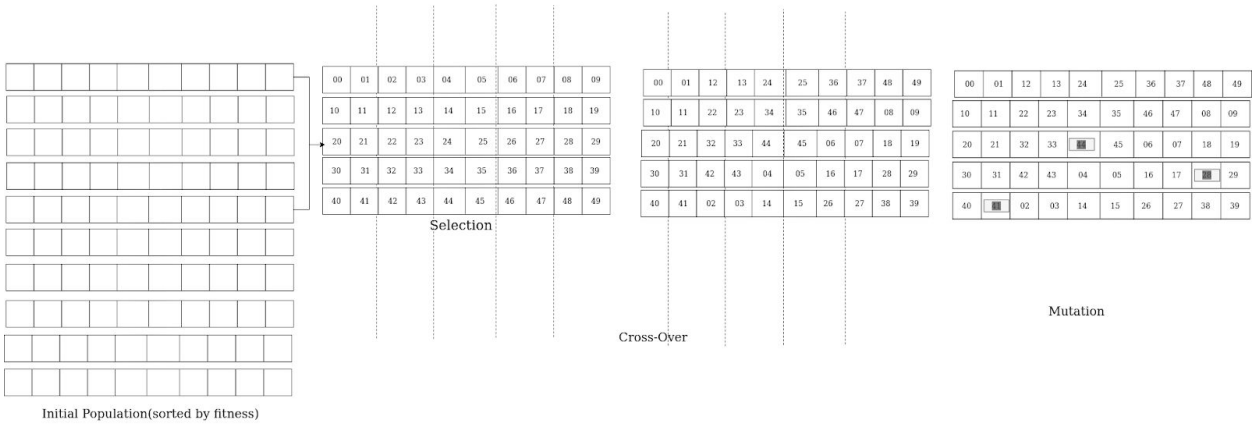
Top 5 chromosomes with best fitness are selected in selection.


00 01 12 13 24 25 36 37 48 49
10 11 22 23 34 35 46 47 08 09
20 21 32 33 44 45 06 07 18 19
30 31 42 43 04 05 16 17 28 29
40 41 02 03 14 15 26 27 38 39

Cross-Over

00	01	12	13	24	25	36	37	48	49
10	11	22	23	34	35	46	47	08	09
20	21	32	33	44	45	06	07	18	19
30	31	42	43	04	05	16	17	28	29
40	41	02	03	14	15	26	27	38	39

Mutation



Fitness function

```
def Calc_initialPopulatin(Error_initial_population):
   for it in range(soln_per_population):

Error_initial_population.append(server.get_errors(key,initial_population[it].tolist()))
       Error_initial_population[it].append(it)

Error_initial_population[it].append(Error_initial_population[it][0]*3.5+Error_initial_population[it][1]*6.5)
   Error_initial_population=np.array(Error_initial_population)
   return Error_initial_population
```

As given in the task, fitness function should be formulated by querying the training and validation sets.

So, we query all the chromosomes in the population to get their corresponding training and validation errors using **get_errors** method. We append those errors to their respective vectors.

Fitness function: sum of validation and training error(with some coefficients).

Now we calculate fitness for every vector and append them to their respective vectors.

The fitness function is selected in this way because the goal is to generate a good fit,so we need to get both errors almost equal, and the population used has almost equal

errors so we need to give equal importance for both errors, but can slightly compromise over train error as given was an overfit.

Crossover Function

```
def Crossover(parents_from_initial_population):
   first_generation=[]
   for i in range(int(soln_per_population/2)):
       temp=[]
       for j in range(num_coefficientWeights):

temp.append(initial_population[int(parents_from_initial_population[i][2])]
[j])
       first_generation.append(temp)


   for i in range(int(soln_per_population/2)):
       temp=[]
       temp.append(-10)
       for j in range(2):
           temp.append(first_generation[i][j+1])
       for j in range(2):
           temp.append(first_generation[(i+1)%5][j+3])
       for j in range(2):
           temp.append(first_generation[(i+2)%5][j+5])
```

```
        for j in range(2):
            temp.append(first_generation[(i+3)%5][j+7])
        for j in range(2):
            temp.append(first_generation[(i+4)%5][j+9])
        first_generation.append(temp)
    return first_generation
```

The initial crossover function used is **Uniform Crossover,** but later as initial population becomes better and better we finally applied is **Diagonal multi-parent crossover**, so that the upcoming results will be diverged and different.

IMPLEMENTATION OF DIAGONAL MULTI PARENT CROSSOVER

Here, we start by taking an empty dictionary and append half the population.

This half consists of the vectors having better fitness(i.e the first half of the population sorted by fitness). Now the remaining half is formed by the following method.

Two consecutive genes are picked from every chromosome(not repetitive for the same weight of vector) and give birth to a new chromosome.

This is repeated in a circular function to populate the remaining half of the population. This is called Diagonal multi-parent crossover.

Now the new set of children chromosomes is appended to its former parents(dictionary).

Thus a new set of population is formed.

Mutations

```
def Mutation(initial_population):

   num=random.randint(4,6)

   i=0

   for i in range(num):

       a=random.randint(1,9)

       b=random.randint(1,10)


initial_population[a][b]=(initial_population[a][b]+(initial_population[a][
b]*random.randint(-0.3,0.3)))%10

   return initial_population

   return initial_population
```

Here, we generate a random number which decides the number of mutations to be applied, where the value will be increased or decreased by 30%. We considered this mutation since we observed that the initial population vector is very sensitive where we land up with huge differences on changing their values.So we tried to bound it.

Hyperparameters

Pool size:

We selected our pool size to be 10 because of the vector size.

Since a vector has 10 features(and so 10 coefficients omitting the first element since its contribution is least) and we are given an overfit vector as reference.We optimised each coefficient at a time and got 10 vectors, so we started with these 10 vectors as initial population. And every time we get better results we update the initial population with these new generated best vectors.

So our pool size is $10 \times 10 = 100$.

Splitting point:

Since we are using diagonal multi-parent crossover function, we had to pick 4 points(5 parts) to have 5 children. We chose to distribute those 4 points evenly in the chromosome by fixing the points after every two chromosomes.

There are two reasons to expect that the use of more parents in diagonal crossover leads to improved GA performance: a high level of disruption and a large sample of the search space used when creating offspring. As for the first aspect, by using more crossover points the operator becomes more disruptive, thus more explorative and less sensitive for premature convergence. Secondly, by the use of more parents there is more information on the search space and there is more consensus needed to focus the search to a certain region, that is the danger of (too) early commitment is reduced.On using this method we have higher probability of not falling into local best, and can achieve global best.

Process / Path we followed in reaching our current best and its Heuristics

Analysis of given Overfit vector and Generating Initial population:

First we analyzed the given overfit vector making some changes in each weight. Then we considered the best out of them as our initial population.So my First generated initial population consists of 10 vectors where each vector has a slight difference with one of the weights of given overfit vector.(The weight of first coefficient remains same as of overfit vector since we didn't observed much changes on changing it)

Initial Fitness Function:

After generating our initial population ,we started on calculating **fitness function** . Our first fitness function was addition of the two errors. (i.e Train_Error+Validation_Error). We observed that errors were decreasing to some extent but not much.

We noticed that the train errors are considerably low, but validation errors are huge..

Crossover and Mutations:

The crossovers we tried are uniform crossover and diagonal multi-parent crossover considering we get a wide range of results.

We randomly selected some positions in the array and replace it with a random number%10

Updating Fitness Function:

Then we started updating our fitness functions in such a way that we decrease our validation error. Then we tried different variants but best one was:

(Train_Error*(X) + Validation_Error*(Y))

We tried different values of X and Y and noticed that the best results are obtained when X<=4 and Y>=6 and X+Y=10

Then we observed some good results where both train and validation are almost equal and are in a range of (10^7).

Then I manually crossovered these best results and mutated them, and the errors are decreased to a range of (7* 10^6)

Mutating the best results:

After the best possible results are achieved we thought of mutating them. Till now our mutation was updating the value, now we thought of just updating the power (or) exponential part.So we tried and analyzed different values as exponents and decided some ranges for each weight .

```
def Updating_exponents(entered):

   i=0

   for i in range(7):

       exp = random.randint(min(-(i*(i-1)), -(4*(i+1))),max(-(i*(i-1)),
-(4*(i+1))))

       if(exp<-13):

           exp=random.randint(-15,-12)

       entered[i+4]=(entered[i+4]*(10**exp))

   return entered
```

Here we didn't performed any crossover but just tried updating the best results we had, and we resulted with even better arrays, and were able to reduce the train and validation errors to the range of (3*10^6).Then on manual changes we were able to reduce it a little.

Statistical Information

1)On an average our vectors converge on 25~35 iterations based on mutations and crossover applied.

We hope that our current **best vector** is almost near to this:

[-10, 3.6732405419386582, -0.09, 0.054, -3.86000157715883e-07, 2.901944449494161e-07, -6.93409e-09, -6.00985565299179e-10, 3.488006383229681e-12, 4.551492499340711e-14, -3.430100176902565e-15]

Because of the errors we got, Train error:259295.9197084011 and Validation error:262101.07415512315

Here the two errors are almost equal which ensures that I am not overfitting the dataset and also we need is the best fit out of given overfit.As given is overfit it performs well on the training data but not on the validation data so we need to give importance for reducing the validation error compromising training error upto an extent as already it's a better one.And here we reduced validation error from 10^8 to 10^6 compromising training error to some extent.

THE ITERATION DIAGRAMS AND TRACE OUTPUTS ARE IN SEPARATE FILES NAMED “IterationDiagrams.pdf” and we have 2 trace outputs “trace1.txt” contains our intermediate best initial population and “trace2.txt” contains our final best initial population.