Machine Data And Learning Project

Team 7 (Pratham Gupta, Arth Raj)

Q1. Summary of the Genetic Algorithm with all the steps, also mention if you have made any major changes to the base genetic algorithm

Genetic algorithms are based on a comparison between the genetic structure and activity of population chromosomes.

Some key terms of Genetic Algorithm:-

- **Individual**: This represents a particular solution to the problem, in our case it is the vector of weights of size 11.
- **Population**: This represents the current set of solutions or individuals
- **Fitness Score**: Fitness score defines the competitive level of a particular solution i.e. how fit the genome is to mate with other genomes.

Operators used in the algorithm:-

- **Selection Operator**: The idea is to give individuals with high fitness scores a leg up and encourage them to pass on their genes to future generations.
- **Crossover Operator**: Individual mating is described by this. A selection operator is used to choose two individuals from the population, and crossover sites are selected at random. The genes at these crossover sites are then swapped, resulting in the development of a completely new organism (offspring).
- **Mutation Operator**: The key concept is to introduce random genes into offspring in order to preserve population diversity and prevent premature convergence.

Algorithm:

First we create the initial population (if initial population already exists from the previous run, we take that one). **Case if code is run for the first time i.e. No previously trained population exists, then:-** We have an individual in the list named original and create the population from it using this code which calls the createPopulation function.

```
population_size=7
population_arr=[]
for i in range(population_size):
    newvec=createGenome(initvector)
    x=Individual(get_errors(SECRET_KEY,newvec),newvec)
```

```
    x.calculateFitness()
    population_arr.append((x,x.getfitness()))
```

This is the **createGenome** function in which a new individual is created by adding a small random constant to the overfit values.

```
newvec=[]
for i in range(len(vec)):
    newvec.append(vec[i]+np.random.uniform(-vec[i]/10000,vec[i]/10000))
return newvec
```

As we create a random individual, we make an object of **Individual** class, where we pass the train and validation errors for the respective vector and then get the fitness value calculated for that vector.

```
def __init__(self,errarr,vec):
        self.__trainerror=errarr[0]
        self.__validationerror=errarr[1]
        self.__weightvector=vec
```

This is our fitness function.

```
def calculateFitness(self):
        self.__fitness=1/(self.__trainerror+self.__validationerror)
```

So now, a **population** is created in which every element is an object of class **Individual**. Now we perform the **Selection Operator**. First we sort the population according to the fitness value (from more fitness value to less fitness value).

```
population_arr.sort(key=lambda x:x[1],reverse=True)
```

In the above population_arr, It is a list of tuples where the first element of tuple if the Individual Object and the second element is the fitness of that particular Individual. Now we **select** best four vectors from the above population_arr, which will undergo crossover for the next generation. The remaining three vectors for the next generation are the three best vectors from the previous generation.

```
childrenarr=[]
    for i in range(4):
        childrenarr.append(population_arr[i][0].getweightvector())
population_arr1=[]
    for i in range(3):
        population_arr1.append(population_arr[i])
```

Now we perform **Crossover(Mating)** We perform crossover pairwise on four best vectors of current generation. We select two random indexes and send them to **mate()** function for single point crossover. Crossover function

```
nextgeneration=[]
s1=random.randint(0,3)
s2=s1
while s1==s2:
    s2=random.randint(0,3)
c1,c2,cc1=mate(childrenarr[s1],childrenarr[s2])
nextgeneration.append(c1)
nextgeneration.append(c2)
arr=[i for i in range(4)]
arr.remove(s1)
arr.remove(s2)
c1,c2,cc2=mate(childrenarr[arr[0]],childrenarr[arr[1]])
nextgeneration.append(c1)
nextgeneration.append(c2)
ccs=[]
ccs.append(cc1[0])
ccs.append(cc1[1])
ccs.append(cc2[0])
ccs.append(cc2[1])
return nextgeneration,ccs
```

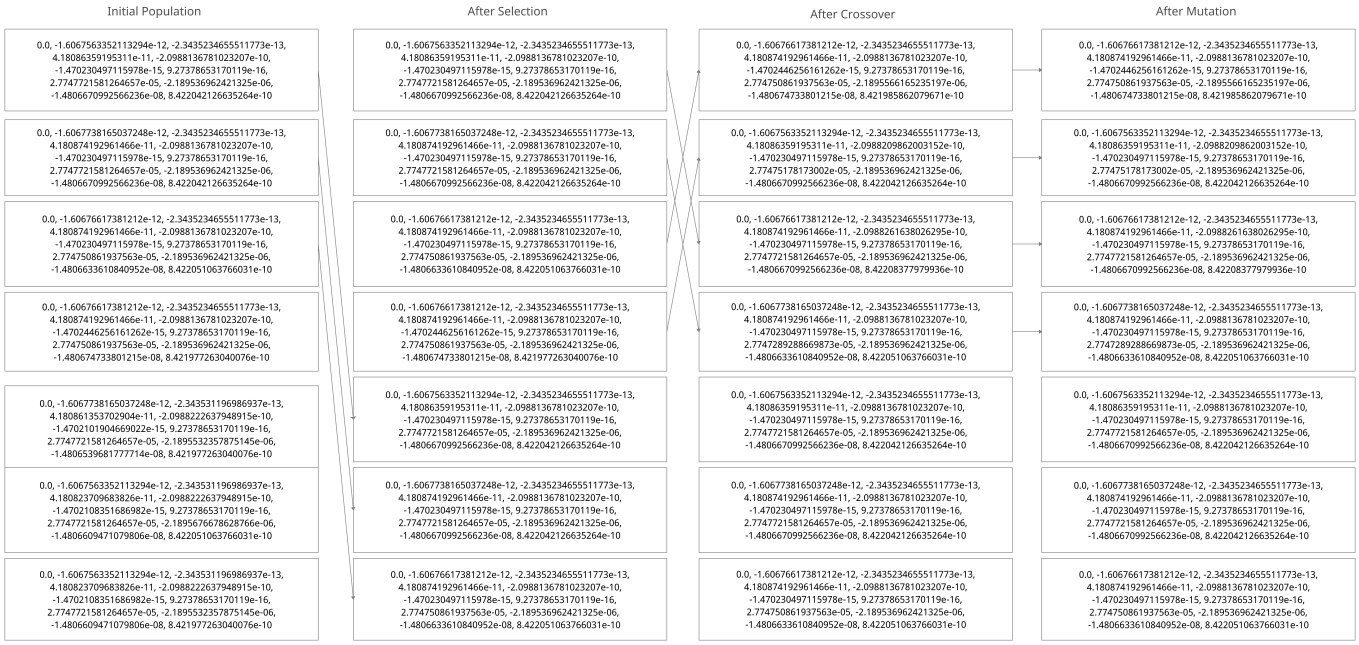
mate() function

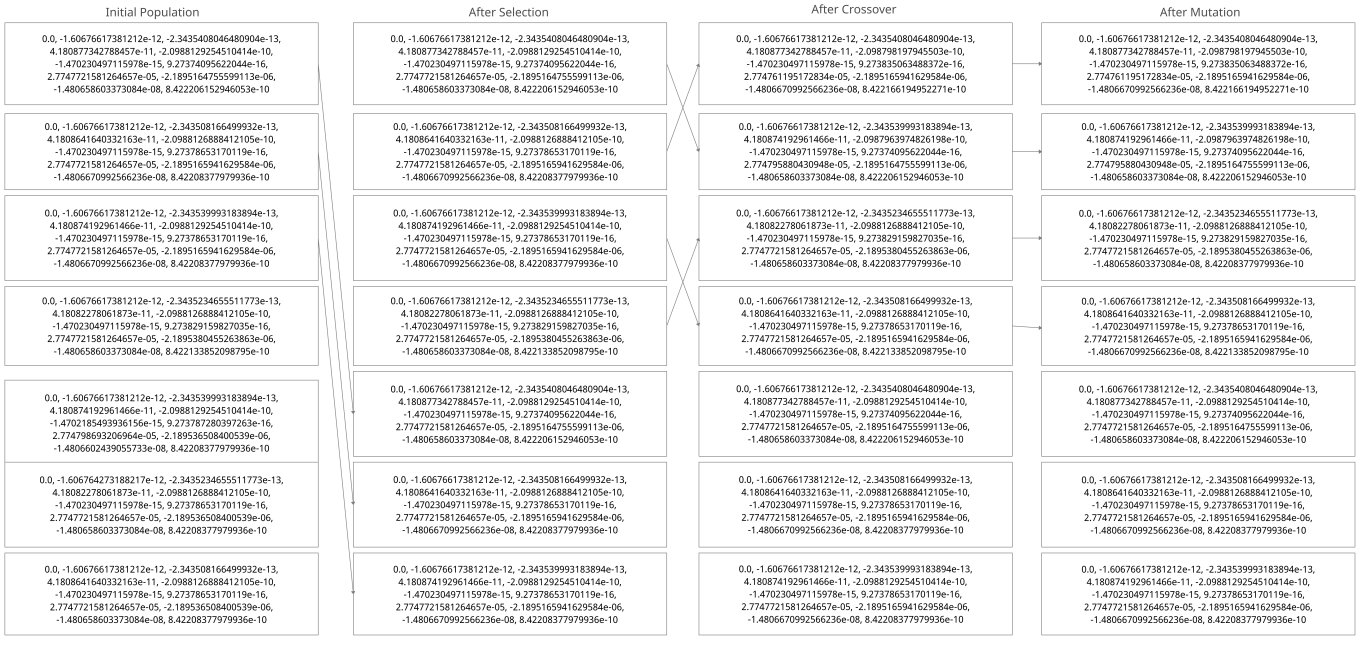
```
child1=[]
child2=[]
sp=random.randint(0,len(v1)-1)
for i in range(sp+1):
    child1.append(v2[i])
    child2.append(v1[i])
for i in range(sp+1,len(v1)):
    child1.append(v1[i])
    child2.append(v2[i])
crosschilds=[child1,child2]
for c in [child1,child2]:
    performmutation(c)
return child1,child2,crosschilds
```

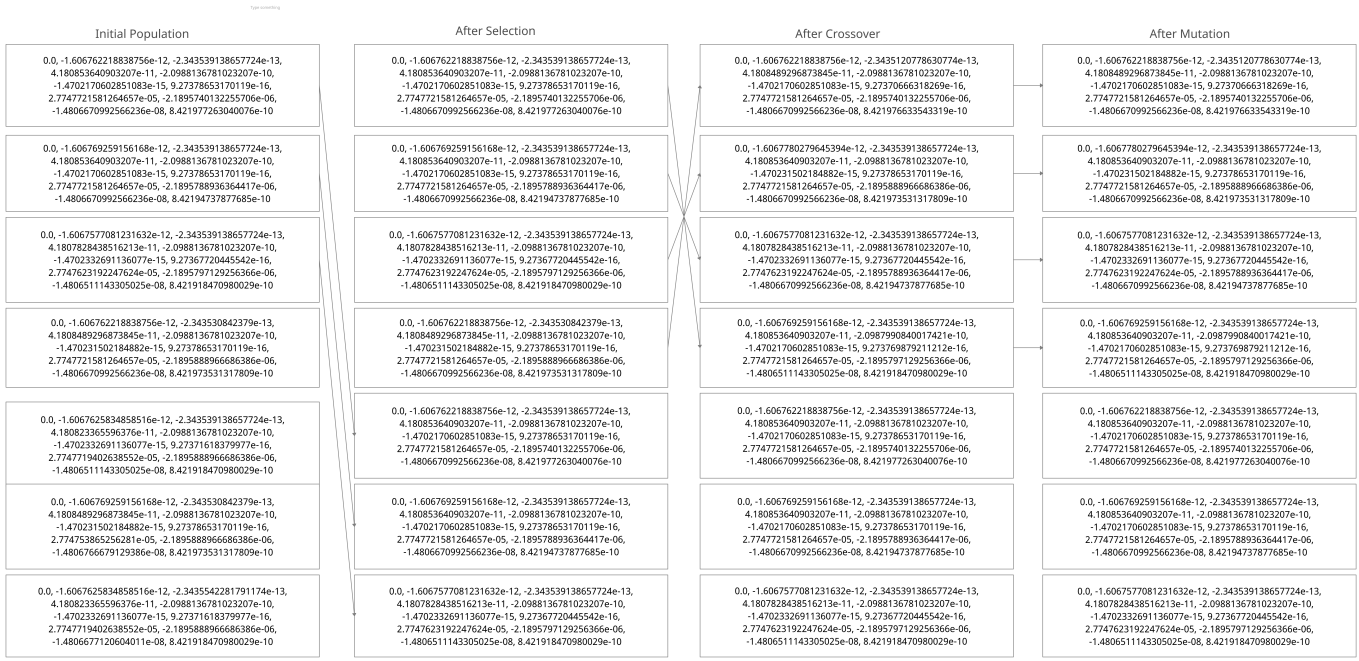
After doing crossover we do **mutation**. For every weight in the vector, we generate a value which tells the probablity whether the weight has to be mutated or not. If generated number is greater than the probablity(80%) then the weight gets mutated by addition of a small value not greater than 1/10th of the original weight. **perform_mutation** function

```
for i in range(len(nextgen)):
    probablity=random.randint(0,100)
    if probablity>=80:
        nextgen[i]+=random.uniform(-nextgen[i]/100000,nextgen[i]/100000)
return nextgen
```

Q2. Three diagrams that showcase three iterations of your algorithm.







Q3. Explain your fitness function.

Initially we used **sum of validation error and test error** as our fitness function, the validation error is minimum , but if take only validation error , we can't compare two individuals having same validation error, so we introduced training error to overcome the issue and now we can notice that if the sum of both validation error and training error is minimum , then our solution is around the optimal point. But later on we took **1/sum** (and reversed the sorting condition) as the sum was going too large due to the computer's inability to store and compare big numbers at a greater precision (5.88888888e120, 5.888888881e120 will be rounded and compared to be equal), we used this as it does the same thing as (x+y) , but now computers can store these numbers.

Q4. Explain your crossover function.

We did single-point random crossover . (A single crossover point is selected along the length of the string at random. The bits lying on alternate sides are merged.)

```
sp=random.randint(0,len(v1)-1)
for i in range(sp+1):
    child1.append(v2[i])
    child2.append(v1[i])
for i in range(sp+1,len(v1)):
    child1.append(v1[i])
    child2.append(v2[i])
```

We are taking a random point lying in the range 0 and 10 and then merging the alternate sides of the two vectors.

Q5. How exactly did you apply the mutations(if any).

For every weight in the vector, we generate a value which tells the probablity whether the weight has to be mutated or not. If generated number is greater than the probablity(80%) then the weight gets mutated by

addition of a small value not greater than 1/10th of the original weight and not less than -1/10th of the original weight.

```
def performmutation(nextgen):
    for i in range(len(nextgen)):
        probablity=random.randint(0,100)
        if probablity>=80:
            nextgen[i]+=random.uniform(-
nextgen[i]/100000,nextgen[i]/100000)

    return nextgen
```

Q6. What were your hyperparameters like pool size, splitting point for creating new genes, etc and why did you choose those parameters?

Pool Size :- First we took pool size as 16 but later on reduced it to 7 since we observed that taking a larger population is in vain as the least fit individuals were never producing the best individuals (better generalised) weights and thus was computationally inefficient to have larger population size.

Mutation Probablity:- First we took the probablity to be 50 percent and observed substantial improvement in our training and validation errors, but later on the probablity has to be increased to 20 percent as after a point of time generations got corrupted owing to previous high probablity of 50 percent.

Q7. Statistical information like number of iterations to converge, etc.

We observed that the last indexed weights are more sensitive upon change, so we applied mutation considering the order of the weights.We could not define a condition for convergence because we did not have the optimal value of the validation or test error. Also we could not use any relation between train error and validation error to be a convergence condition due to high fluctuation in their values. So we were running the algorithm random number of time and random number of iterations to get a considerably generalised model.

Q8. What heuristics did you apply, mention the ones that didn't work too.

The following are the heuristics that are applied while trying to generalise the model better.

- **Weight Sensitivity**:- We observed that the last indexed weights are more sensitive upon change, so we applied mutation considering the order of the weights.
- **Fitness Functions**:- Here x=> train error, y=> validation error.
    - **(x+y)**:-The above mentioned functions were not producing better results ,the reason being , if (y-x) is small , it doesn't imply an optimal point, we can see that if the model is highly underfit, then also (y-x) is small. so we tried other functions.
    - **1/(x+y) (Final)**:- This produced the best result as values of the range 1e-12 can easily be compared.
- **Modifying the mutation function** :-
    - Initially we were randomly selecting the number in the range (-10,10) which was producing very bad output.

- Then,later on we added a small random constant in the current gene (bit),which was somewhat better.
- But later on we realized that the small random constant too can be very large depending on the order of the current gene,so calculated the order of the current gene and the added the random number of that order only

Q9. Mention the train error and validation error that you were able to achieve. Explain why this performs well on an unseen data with theoretically sound points. Avoid vague arguments.

Train Error:- 1.4*1e13

Validation Error:- 6*1e12

We are not certain if this performs well on unseen data, but if the data highly randomized at your end, we might end up getting good results with our best vector.