MDLProject

March 17, 2021

1 Genetic Algorithm

Team 39 : Moodles Team Members: 1. Rajat Kataria (2019101020) 2. Tanishq Goel (2019114015)

1.1 Summary

Initial_vector =

```
[0.0,
-1.45799022e-12,
-2.28980078e-13,
 4.62010753e-11,
-1.75214813e-10,
-1.83669770e-15,
 8.52944060e-16,
 2.29423303e-05,
-2.04721003e-06,
-1.59792834e-08,
 9.98214034e-10]
```

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection.

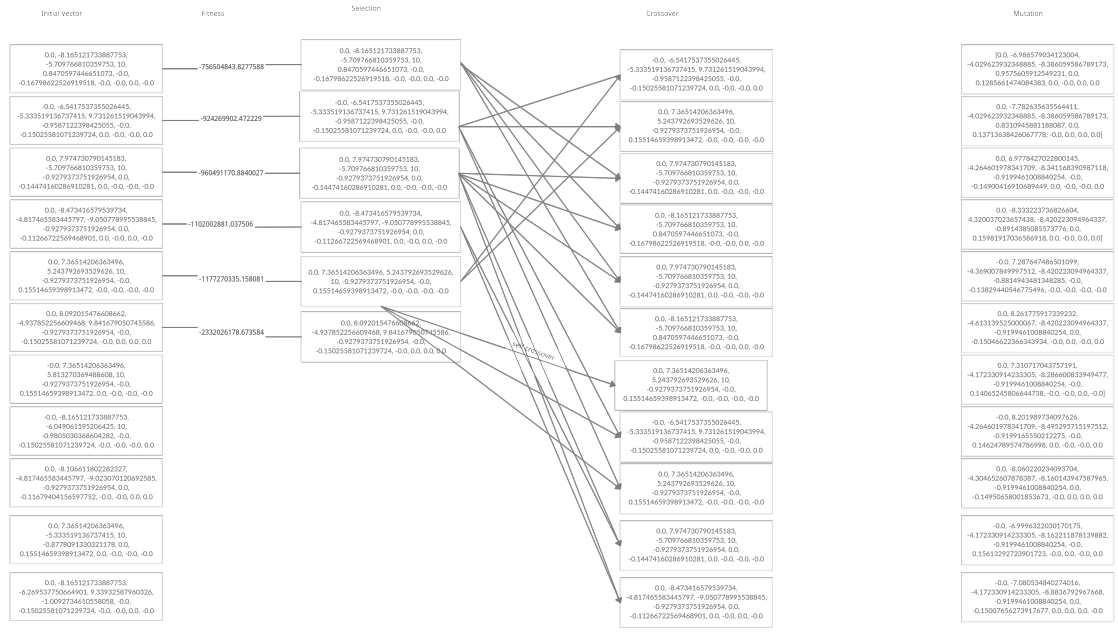
We were given an overfit vector initially whose { validation_error / training_error ~ 10 }. From the initial vector we made a pool or population of possible solution by taking the given vector and generation 10 more vector from that given vector using mutation as which is explained later. The best fitest from these solution then undergo recombination and mutation for every generation producing new vectors which then undergo same process of selection -> crossover -> mutation. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. This is in line with the Darwinian Theory of "Survival of the Fittest". In this way we keep, evolving better individuals or solutions over generations, till we reach a stopping criterion . Here we limited the number of generation to match the intermediate solution for best fitness because the algorithm was randomized .

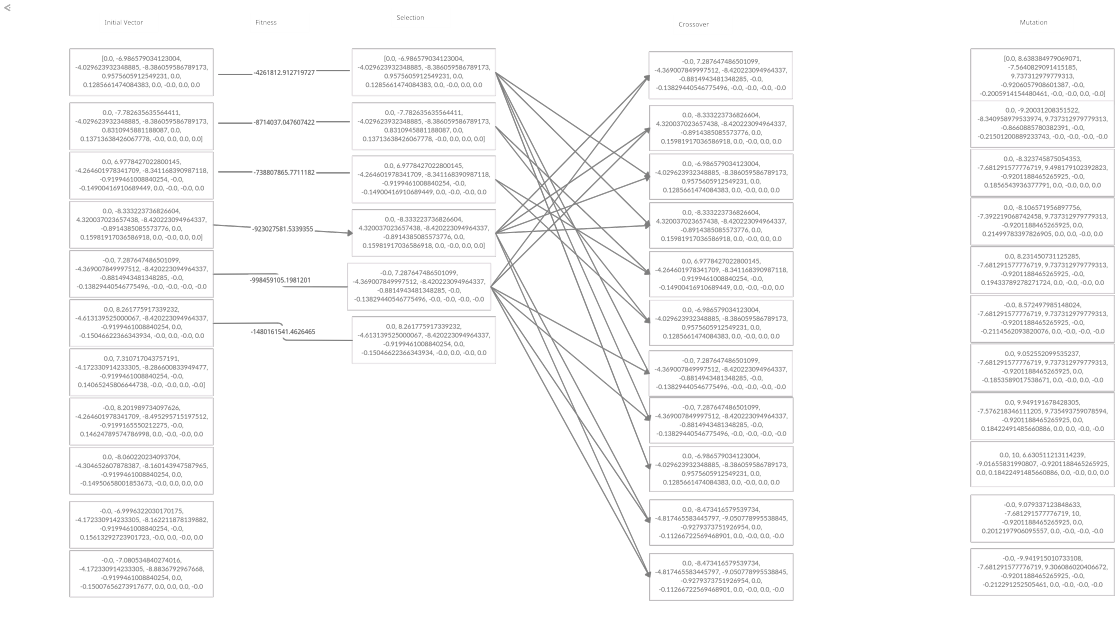
We also tried various random solution from the vector which we were getting keeping track of the best so far and making the generation again from that vector in order to achieve the fittest.

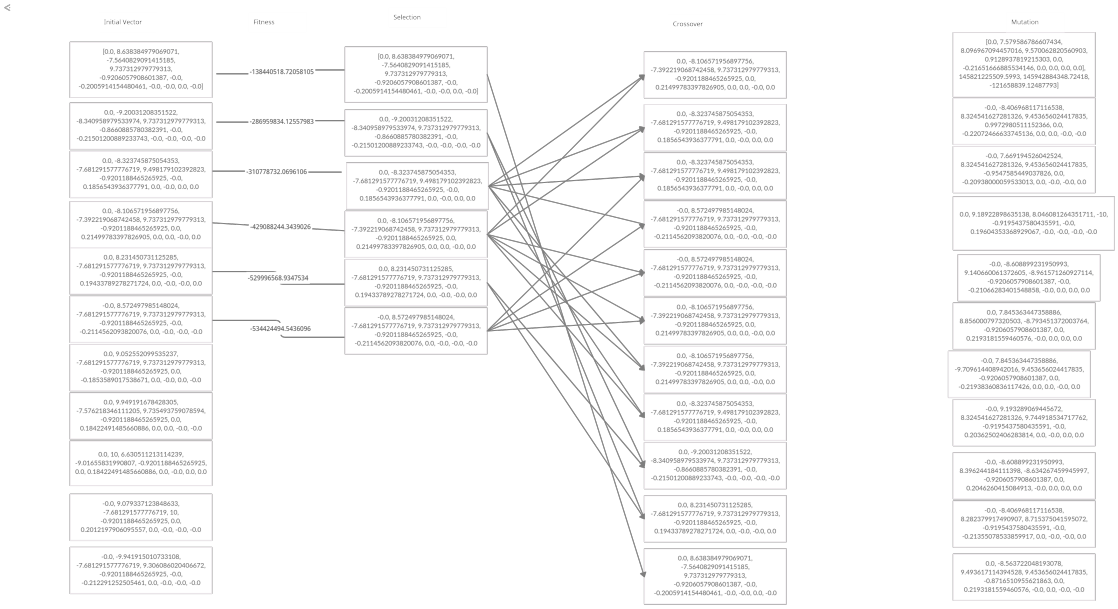
We kept changing the fitness function in order to divert the vector in the direction which we want it to. Overview of algo applied -

1. Initialize a population
2. Untill number of generation repeat following:
    1. Determine fitness of population
    2. Select parents from the population and crossover them
    3. Perform mutation on new population

Every individual is a vector of size 11 with 11 floating point values in the range of [-10, 10].







1.2 Fitness Function

Fitness function returns the fitness of a particular vector . The value of this fitness as compared to other vector decides whether will this vector take part in crossover for producing new vectors or not. We kept changing the fitness function in order to check which fitness function is resulting in selection of best parents which produce chilldren with less MSE and good rank on leaderboard . Firstly we selected the fitness function as

```
def errorToFitness(train_err,validation_err):
return -(X*train_err + Y*validation_err)
```

Here X is the train factor and Y is the validation factor . We kept changing the train factor and validation factor and initially kept that 0.7 : 1 to get rid of overfit . The initial vector had very low train error and more validation error. With our fitness function, we associated less weight to the train 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 0.6 and 0.5 in the middle to get rid of the overfit faster. 0.7, however, worked best as it did not cause train error to rise up suddenly, unlike its lower values.

We also kept fitness funtion function as

```
def errorToFitness(train_err,validation_err):
    return -abs(train_err - validation_err)
```

This was done in order to decrease the difference between train_error and validation error . As it selects the vector whose difference between the train error and validation error is nearly same.This will lead to decrease the difference between train error and validation error.

1.3 Crossover Function

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. This includes self crossover as well as 2 vector crossover . The crossover point was selected at random using np.random.randint(CHROMOSOME_SIZE) 'create_mating_pool' function generated the mating pool which means picks up top MATING_POOL_SIZE number of vectors.

```
def create_mating_pool(population_fitness : list):
    mating_pool = population_fitness[:MATING_POOL_SIZE]
    return mating_pool
```

Basically, the first parent was copied till a certain index, and the remaining was copied from the second parent. Similarly, for second child.

```
def crossover( mom : list, dad : list):
    thresh = np.random.randint(CHROMOSOME_SIZE)
    alice = mom.copy()
    bob = dad.copy()
    alice[0:thresh] = mom[0:thresh]
    bob[0:thresh] = dad[0:thresh]
    return alice,bob


def normal_crossover(mom : list, dad : list):
    thresh = np.random.randint(CHROMOSOME_SIZE)
    child = dad.copy()
    child[0:thresh] = mom[0:thresh]
    return child
```

We made two function for crossover one would return just one child and other would return 2 childs formed by the vectors. Here we can see

```
thresh = np.random.randint(CHROMOSOME_SIZE)  # thresh is the crossover point which is decided ra
```

1.4 Mutations

Mutation is a genetic operator that changes one or more gene values in a chromosome. Therefore, we implemented it in a way that it will mutate every gene of an individual which is 11-D vector here with a probability of MUTATE_PROB. We also kept in view about to change a value if the value is 0. Initially we used to change the value from 0 to random(-0.01,0.01). Changing this just increased the error in comming generation,so we decided to remove it from code as we think making a coefficient 0 decreases the overfit and MSE and makes the model simple

```
def mutate(mutate_vec):

    for i in range(len(mutate_vec)):
        for j in range(11):
            if np.random.uniform(0,1) < MUTATE_PROB:        # MUTATE_PROB = 4/11
                mutate_vec[i][j] = mutate_vec[i][j]*np.random.uniform(0.9,1.1)
                if np.random.uniform(-1,+1) < 0:
```

```
                mutate_vec[i][j] *= -1
            if mutate_vec[i][j] > 10:
                mutate_vec[i][j] = 10
            elif mutate_vec[i][j] < -10:
                mutate_vec[i][j] = -10
    return mutate_vec
```

In the above code every value is changed according the mutate probability and is randomly multiplied by any number between (0.9,1.1) as well as -1 or +1.

1.5 Hyperparameters

1.5.1 Population Size

As we were supposed to use limited API calls during the working span of assignment. It was very important to choose a proper population size of a generation. We initially kept it at 8. It was limiting the search space due to less randomness in population. Later when API calls increased we shifted to 15 but finally kept it to a satisfying level of 11.

1.5.2 Mating Pool

Initially we kept mating pool size to 4 and logic was such that next generation contains 4 parents and 7 crossed-mutated children from the mating pool. But that lead to overfit condition. We relied it as our vectors error got optimized to order ~11 but rank on the leader board fall down to 80-90. That obviously represent the case of overfit as our model performed well on train & validation data but not on test data. Finally we kept mating pool size to same as `POPULATION_SIZE` and logic was such that next generation contains only crossed-mutated children from the mating pool. We got an error of order 12 which was not as previous one but our rank improved to 50 that meant it is not overfit now and will perform well on testing data.Moreover we finally included the passing of parents in the future in order to do not lost the best vector.

1.5.3 Cross-Over Point

We wanted to randomness during cross-over as much as possible so instead of choosing a fixed index as a cross-over we kept it a random number between 1 to 11 both including.

1.5.4 Mutation Factor

Initially we kept our mutation multiplication factor to be in range (0.009,1.001) such that we don't lost current generation fitness much and at same point bring some slight variation in the population, but only for lower order coefficients such that we can explore much deviation for lower order at time of convergence and kept long range(0,1) for higher order for making start with greater variation for more randomness in generation's vectors. Later we changed (0.9, 1.1) to (0.7, 1.5) and (0.1,1.9) while doing hit & trial we found errors to increased to order 15. hence discarded later changes. As we were increasing the values variation it helped us once while we were stuck in a local minima but later on high mutation was giving us random error and vetor so we decided to keep it (0.9,1.1) so as to do slight change and api's request were also increased so we could run for more generations.

1.6 Statistics

We made our code run for fixed number of generations i.e. `NUMBER_OF_GENERATION`. This was because of limited daily API calls. But after running till 30th generation we got stuck with fixed error(both train & validation) of order 1.02*e^11. We reasoned for that to be stuck in local minima. Even after doing some changes in mutation factor as described in Hyperparameters section we didn't get much improvement. So we decided to bring in some randomization in our initial vector. And applying regularisation with it using data generated earlier. After that we can say running a loop till `~60` generation will give the optimized vectors with best errors globally.

1.7 Heuristics

1. We took the mutation probability to 8/11 which didn't work for us as it caused a lot of variation in the complete data which was just driving us to an vague result. Other mutation probability which we tried were 6/11, 4/11, 2/11, 3/11 but the best result which we got was by 4/11. We think this was because the main parameters on whom our model best fits are from 4 to 6 number of parameters. The other degree is just causing overfit.
2. We took different fitness function which is told above but the best fitness function which fit were train == train_error || valid == Validation_error
    - f(X,Y) = -(X*train* + *Y*valid)
    - f = -abs(train-valid)
    - f = 1/(X*train+valid)
    - f = 1/(X*train - valid)
3. We tried different mating pool size:
    - mating pool = 4 This caused a lot of focus on a particular direction and decreased randomization
    - mating pool = 8 This caused a lot of randomization and was not directing result to a particualer minima or maxima
    - mating pool = 6 We found this as the best fit mating pool size as it also took best fit with some randomized vectors in it

1.8 Errors

- Generation : 60 - 75 (If the mutation probability and radomization is set to right values and fitness is wisely selected)
- Vector : [0.0, 9.18922898635138, 8.046081264351711, -10, -0.9195437580435591, -0.0, 0.19604353368929067, -0.0, -0.0, -0.0, -0.0]

- Train Error : 145779841785.92172
- Validation Error : 146147729531.04205
- Theoretical validation argument : We think this vector is giving minimum error in train and validation as well as a good rank because our model complexity is of degree 5 to 7 but we were given a vector of degree 11 which was causing overfitting. Moreover overfitting error on train and validation were near to ~ 10^10 to ~10^11 .So we think error of order 10^11 to 10^13 is a good thing as to reduce overfitting bias would have increased but variance would have decreased as compared to overfit once. If the errors are greater than 10^13 ,we will

consider it as an underfit model which will not perform well on test data and will decrease rank on leaderboard.

1.9 Others

- We used regularization by decreasing some values in vector nearly to 0 in order to reduce complexity and to prevent overfitting.
- We used to take a random vector from between the list of vectors and start the process from starting in order to reduce the historical randomness among them and to make randomness among vectors that we wanted.