MDL PROJECT: GENETIC ALGORITHMS

Team 95: Arushi Mittal (2019101120) Meghna Mishra (2019111030)

Project Description

```
Using the given coefficients of an overfit model, apply genetic algorithms in order
to generalize the model for a variety of datasets. Vectors consisting of 11 floating
numbers in the range [-10, 10] are used to represent individual members of a
population. The submission contains 10 top vectors which contain the coefficients of
the individual features and can reduce the overfitting on the given population.
```

Genetic Algorithms

```
Genetic Algorithms are based on Charles Darwin's theory of natural evolution which
states that the process of natural selection ensures the survival of the fittest
members of a population which over time, leads to an overall improvement in the
characteristics and abilities of all subsequent populations. Genetic algorithms are
metaheuristics inspired by this theory that attempt to simulate the processes of
selection and genetics. These algorithms are randomized search algorithms and are
used to solve optimization problems so that the problems and solutions can evolve,
especially when an objective function value needs to be maximized or minimized under
some given constraints.

The following are the phases of a genetic algorithm:
1. Initial Population
2. Fitness Function
3. Selection
4. Cross Over
5. Mutation
```

Initial Population

In genetic algorithms, the initial population is used to create subsequent generations so that the model can be improved over time. This initial population contains members who can be used to create better children who can in turn be used to create better children. For this model, the initial population we used multiple approaches. The first approach involved using the initial vector, which led to high validation errors. The second approach we used involved using a vector of random numbers which led to some very mixed results, so because of the unpredictability we had to let go of it. The third approach involved mutating (described in the section on mutation below) the original vector by a small factor and small probability which led to issues similar to those of the original vector. Using a zero vector led to very low errors but it was not a good fit generally. That led to our final approach which involved mutating the initial vector with a high probability and low factor so that most of the original vector's components would change by a small amount. This ensured we were following the right approach but made minor changes so the vector was not overfit.

```
for i in range(popsize):
        population[i] = mutate(initial_vector, 0.9, mut_range)
```

Fitness Function

The fitness function is used to determine how compatible a data point is with the model. For this project the fitness function we used varied at different points. Initially we just used the reciprocal of the total error (train error + validation error). After this, we realized that while the total error was decreasing, the absolute difference between the train and validation error values was significant. Therefore we decided to use the reciprocal of the sum of the total error and absolute difference. While this did help, the effect was not significant enough, so we increased the weightage of the difference by multiplying it by 1, various floats between 1 and 2, 2, 5, 10, 50, 75, 100, 150, 200, 500, 1000 and 2000. We finally settled on 2000. We did try using random numbers but the results were not significant enough to warrant continued use of this method.

```
factor = 2000

for i in range(popsize):
    err = client.get_errors(key, population[i].tolist())
    errors[i] = np.copy(err[0]+err[1]+factor*(abs(err[0]-err[1])))
    errors1[i] = np.copy(err[0])
    errors2[i] = np.copy(err[1])

indices = np.copy(np.argsort(errors))
```

Selection Function

The selection function determines which individuals from the population are chosen to be the parents for the next generation. The selection function we have used is relatively simple. It is a modified version of the roulette wheel selection algorithm. Since we are selecting k parents, where k is 6, we pick the top k fittest individuals from the population. These individuals are then given different probabilities depending on their fitness. The probability array determines which individuals have a greater probability of getting selected, so that the children produced are better and can thus make better children for the next generation. As a result, members are chosen solely from the top k vectors.

```
for i in range(popsize):
        population[i] = np.copy(temp_population[indices[i]])
        errors[i] = np.copy(temp_errors[indices[i]])
        errors1[i] = np.copy(temp_errors1[indices[i]])
        errors2[i] = np.copy(temp_errors2[indices[i]])
parents = np.copy(population[:k])
indarr = np.random.choice(k, 2, replace = False, p=[0.3, 0.3, 0.15, 0.1, 0.1, 0.05])
ind1 = indarr[0]
ind2 = indarr[1]
parent1 = np.copy(parents[ind1])
parent2 = np.copy(parents[ind2])
```

Crossover Function

The crossover function describes how the genes of the individual parents are combined in order to produce children. In this case there are 2 children produced by two parents. For the crossover function we used a simulated binary crossover. This involves setting a random number to calculate the distribution factor depending on the value of the random number. The distribution factor determines how close the children are to the parents - larger factors indicate larger differences. The children are both calculated in different ways, where for the first child, the first parent is more significant and for the second child, the second parent is more significant. A total of 10 children are created by crossing over the 6 parents.

```
def crossover(parent1, parent2):
    child1 = np.zeros(11)
    child2 = np.zeros(11)
    parent1 = np.array(parent1)
    parent2 = np.array(parent2)

    n = 3

    u = random.random()
    if(u <= 0.5):
        b = (2*u)**(1/(n + 1))
    else:
        b = (1/(2*(1 - u)))**(1/(n + 1))
```

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

    return child1, child2
```

Mutation

The mutation function involves simulating mutations in the real world, where genes can sometimes be changed due to external circumstances and biological reasons. Here, the mutation involves probability and range. The probability determines how likely a gene is to get mutated and the range determines the extent to which it can get mutated. For the initial vector, we wanted the mutations to occur more frequently in order to generalize the algorithm and we used probability of 0.9 and range of 10%. For the next generations, we used mutation probability of 0.4 and range of 10% so small changes would occur but less frequently. If the number is zero, then a random number on a very small scale is added to the zero data point to induce some mutation but not disrupt the shape of the model.

```
def mutate(arr, prob, mrange):
    for i in range(len(arr)):
        val = np.random.uniform(-mrange, mrange)
        if(arr[i] == 0.00000000e+00):
            val = random.uniform(-1e-11, 1e-11)
            arr[i] = np.random.choice([arr[i] + val, arr[i]], p=[prob, 1-prob])
        else:
            arr[i] = np.random.choice([(arr[i]*val) + arr[i], arr[i]], p=[prob, 1-
prob])
        if(arr[i] > 10):
            arr[i] = 10
        elif(arr[i] < -10):
            arr[i] = -10

    return arr
```

Algorithm and Code Explanation

The algorithm is simple. We create the initial population from the given vector as explained above and then used the get_errors function in order to determine the errors for each member of the population. The fitness function was applied using these error values and then it was used for selection of the parents. The parents were selected using the selection algorithm described above after which the children were created.

```
child1, child2 = crossover(parent1, parent2)
```

After crossover, the children were mutated.

```
child1 = mutate(child1, mut_prob, mut_range)
child2 = mutate(child2, mut_prob, mut_range)
```

If any of the children were duplicates of the parents, they were immediately discarded. This was achieved by creating a while loop that ran until the number of children was 10. If the children were duplicates, the iterator did not update, and the clones were discarded from the populations of crossovers and mutated children.

```
comparison1 = (child1 == parent1)
comparison2 = (child2 == parent1)
comparison3 = (child1 == parent2)
comparison4 = (child2 == parent2)
if(comparison1.all() == True or comparison2.all() == True or comparison3.all() ==
True or comparison4.all() == True):
    gen_crossover_children[-1].pop()
    gen_mutated_children[-1].pop()

    gen_crossover_children[-1].pop()
    gen_mutated_children[-1].pop()

    gen_crossover_children[-1].pop()
    gen_mutated_children[-1].pop()

    gen_crossover_children[-1].pop()
    gen_mutated_children[-1].pop()
    continue
child_population[x2] = child1
x2 += 1
child_population[x2] = child2
x2 += 1
```

After the child_population array has finally been set, we find the errors for the children using the API. The benefit we get from this is that we are able to save our API calls. Since we use 16 calls in the beginning in order to find errors in the population for sorting, we have the errors for the parents. When the children are created, they are all chosen for the next generation along with the set of parents from the previous generations. This way, the number of calls is 16 + 10 * n where n is the number of iterations. This is how the next generation population is determined.

```
for i in range(popsize-k):
    new_population[i] = np.copy(child_population[i])
    new_errors[i] = np.copy(child_errors[i])
    new_errors1[i] = np.copy(child_errors1[i])
    new_errors2[i] = np.copy(child_errors2[i])
for i in range(k):
    new_population[popsize-k+i] = np.copy(population[i])
    new_errors[popsize-k+i] = np.copy(errors[i])
    new_errors1[popsize-k+i] = np.copy(errors1[i])
    new_errors2[popsize-k+i] = np.copy(errors2[i])
```

Hyperparameters

POPULATION SIZE: 16

After trying 10 and 20 as population sizes, we realized that 10 was too restricted and did not yield sufficiently diverse results. On the other hand 20 was too expansive to manage and used a lot of API calls. The results in both were not ideal, and 16 gave us the advantage of both sizes.

SELECTED PARENTS PER GENERATION: 6 Using 6 parents from each generation ensured that the best parents are used to mate so the subsequent generations are better. Additionally, carrying them forward to mate with superior genes ensures the next generation will be even better.

CHILDREN PER GENERATION: 10 Creating almost double the number of parents ensures sufficient diversity and opportunity for mutation along with different combinations of the parent vectors. Additionally, since these children have a high probability of being superior to the previous generations, they will undoubtedly create better offspring and a more diverse pool. This ensures that if the children are mutated too much or not ideal, the parent genes can hopefully offer correction and if not, the subsequent generations are better than the previous ones.

Heuristics

The project was completed by dividing it into two different phases - the minimization of error phase, and the balancing of train and validation error phase. Dividing it this way ensured that the error was sufficiently reduced before we began to generalize for a larger dataset. Since the model works for data with respect to all datasets not just the training data or the leaderboard data, it is generalized for various datasets.

We used various heuristics as explained in the functions above. We tried using various types of initial vectors before we settled on our current strategy of mutating the initial vector by a large probability and a small factor.

We varied the number of parents and children a few times before settling on the current version. This way we were able to settle on our final numbers. The decision to include all the children and the best parents ensured that the subsequent generations showed the greatest improvements. This decision ensured that top vectors from previous generations remained in the next population, while also introducing sufficient diversity in the population. Other approaches with fewer children were too repetitive and yielded no new results, while too many children made the results very erratic.

We varied the factor for absolute difference while calculating fitness so we could bring the train and validation errors as close as possible and settled on 2000 after trying smaller values that barely made an impact.

Initially, we were calculating the errors inside the iteration loop for each population which wasted a lot of API calls. We then switched to the current approach which calculates the initial errors outside the loop and then calculates the errors for children in each iteration. The API calls per run went down from (populationsize + number of children) * number of iterations to population size + (number of children * number of iterations). The extra API calls helped us make better judgements and improve the model with better results everyday.

During error minimization, we encountered a total error value that did not change at all after 20+ iterations, leading to the realization that we have hit a local minima. In order to prevent this, we mutated the top vector the way we mutated the initial vector and progressed from there (by increasing probability of mutation).

Diagrams

Diagram 1

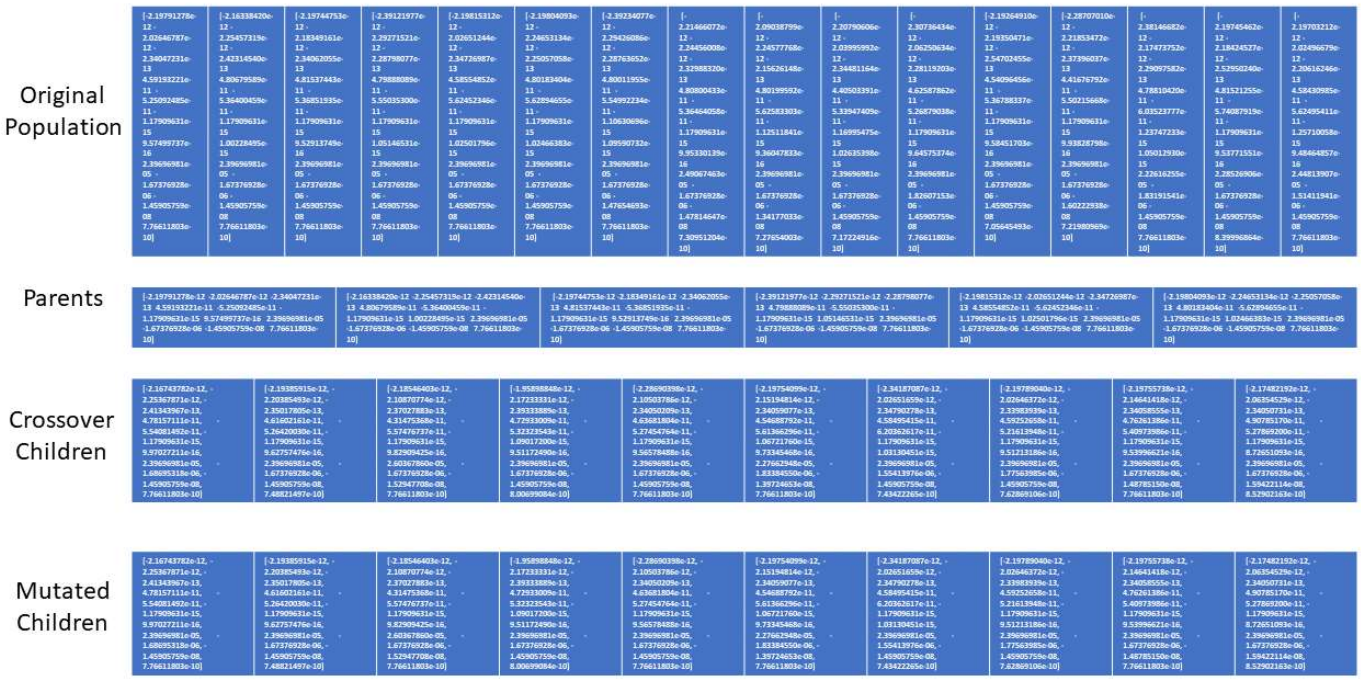
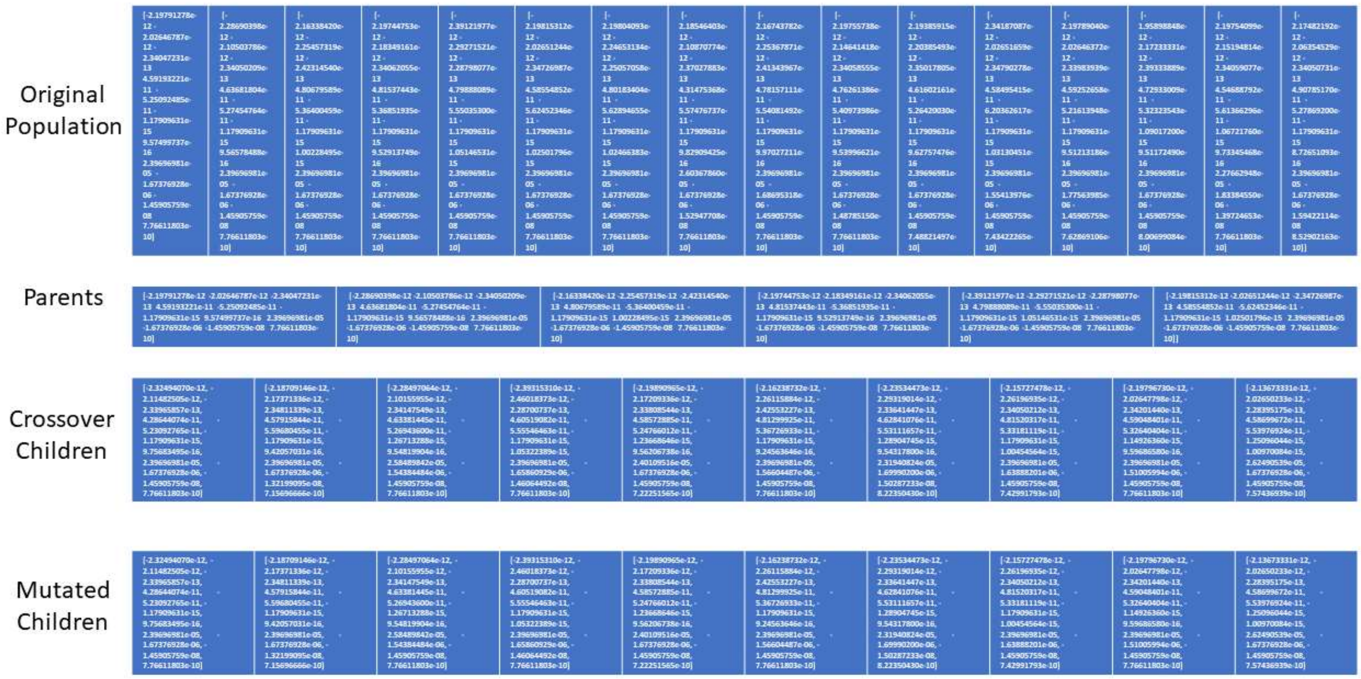
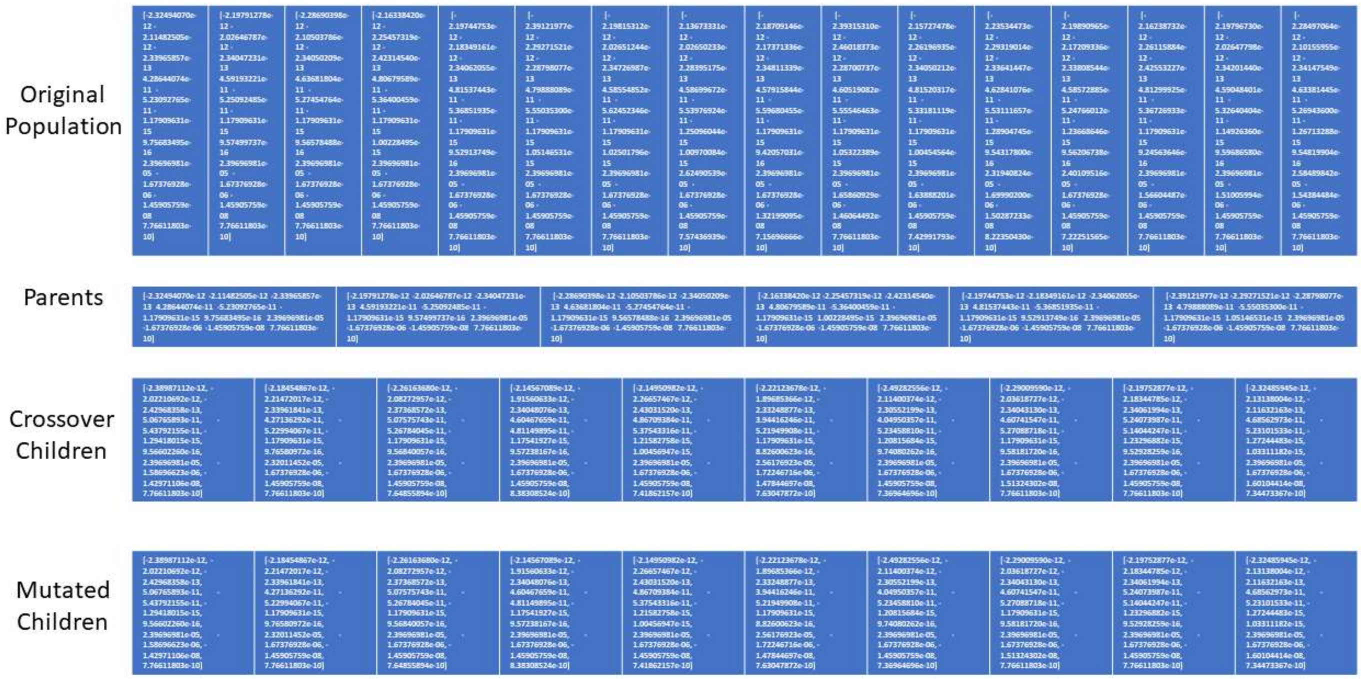


Diagram 2



Original Population
Parents
Crossover Children
Mutated Children

Diagram 3


Original Population
Parents
Crossover Children
Mutated Children

Statistics

The top vectors were chosen on the basis of how much they were able to reduce the train and validation error and subsequently, the total error, along with their ability to reduce the absolute difference between the train error and validation error. The absolute difference between train and validation error is roughly 0.27 percent of the train error. The train error is roughly 49.9 percent of the total error which we believe is a fairly even split.

TOP VECTORS:

[-2.1343305086353493e-12, -2.2896363962565298e-12, -2.255677140167976e-13, 5.097975671687973e-11, -5.174042346093453e-11, -1.1790963051945613e-15, 1.0359193610109647e-15, 2.3969698057747152e-05, -1.6737692764075564e-06, -1.4590575921693652e-08, 7.766118028201611e-10]

TRAIN ERROR: 6.484964039445597E+10
VALIDATION ERROR: 6.467263187502826E+10
TOTAL ERROR: 1.29522E+11
DIFFERENCE IN ERRORS: 1.770085194E+8

We used a total of 8000 - 10 000 API calls in order to generate roughly 400 generations of the initial population.This is exclusive of the datapoints which led to flawed approaches or unacceptable errors.