MDL PROJECT REPORT

Team Number: 19
Team Name: Confused_duo
Team Members:
N.V Sree Harsha Reddy (Roll no:2019101040)
Arumulla Sri Ram (Roll no:2019101032)

Summary of Genetic Algorithm:

We have taken the given overfit vector. We have taken 15 copies of the overfit vector and we have a generated another vector from the copied vector using normal distribution. We have taken some 5 random genes from the normal vector and put them in the copied vector. In this fashion we have generated 15 vectors which serve as the initial population for the genetic algorithm.

We have calculated the fitness values for the chromosomes in the population. We have sorted the chromosomes in decreasing order of their fitness values. We have 10 vectors from the top and store them as ‘top10’ vectors. The ‘top10’ array of vectors contains the best fitted 10 vectors of the whole algorithm and it gets updated after every generation.

With this initial population we start generations. We take the initial population as Parent vectors for the first generation. We sort the parent vectors with their fitness values. We take a fixed(numParentsMating) number of best-fit vectors as the mate-pool. This ensures that the best features of the current generation are passed onto further generations.

Now, we make a fixed number of pairs of chromosomes from the mate-pool. We select each pair of chromosomes based on their probabilities which are proportional to their fitness values, (Roulette algorithm) such that they generate two off springs which contains the traits from both the parents based upon the crossover function. We store all the off springs in an array. Now each gene in the

offspring is subjected to mutation with some probability. We store all the mutated off springs in the child array. We calculate fitness values for each child and sort them in decreasing order of their fitness values. After this we compare the children with the top10 vectors and take the best-fit vectors into the top10 array.

We take some best-fit vectors from the mate pool, and from mutated Childrens. We take some best-fit vectors from the combination of parent vectors (other than the vectors in mate pool) and remaining mutated Childrens (that are not selected in the previous step). We concatenate all these vectors to ensure that the population has enough diversity, and we constraint them to the population size (Population). This concatenated array of vectors forms the Parent array for the next generation. And the generations algo follows.

Fitness Function:

The Fitness Function is calculated on the basis of errors as returned by get_errors function, the errors are **Train Error** and **Validation Error**.

We have **aa, bb** as parameters, the parameter aa describes the weightage that should be given to sum of Train Error and Validation Error, and the parameter bb describes the weightage that should be given to absolute difference of Train Error and Validation Error, we varied these parameters based on the errors we got for submitting the best one.

The denominator of the fitness value is
cc= (aa*SumofErrors + bb*abs(differenceofErrors))

We have taken a constant value 1e15 in numerator to get observable values as fitness values, because the errors are in range of 1e10.

The fitness value is 1e15/cc.

The chromosome which has more fitness value is considered as better, and passed to next generations.

Crossover Function:

The Crossover function takes arguments as all parents and pair of indices for which we are going to perform crossover, these pair of indices are generated by selecting parent's indices based on their probabilities which are proportional to their fitness values, and the function generates off springs and returns them as output. We are using Highly Disruptive Crossover- Heuristic Uniform Crossover (HUX) algorithm. The algorithm as follows:

1. Based on passing pair of indices we are taking the two parents form parents' array which is passed to this function and let's say parents are $A^{(t)}$ and $B^{(t)}$
2. Creating two off springs $C^{(t+1)}$ and $D^{(t+1)}$ as $C^{(t+1)} = A^{(t)}$, $D^{(t+1)} = B^{(t)}$
3. Now we will calculate the no of different genes in $C^{(t+1)}$ and $D^{(t+1)}$
4. We maintain a swap counter, and we will run a while loop this loop stops when the swap counter <= num of different genes/2
5. In while loop we will iterate in the size of length of vector, and in that we are checking an if condition $c_i^{(t+1)} \neq d_i^{(t+1)}$ and $c_i^{(t+1)} \neq b_i^{(t)}$ if this satisfies then we are generating a random number uniformly and checking that the random number is <0.5, if it is then we are swapping bits like this $c_i^{(t+1)} = b_i^{(t)}$, $d_i^{(t+1)} = a_i^{(t)}$ and then increasing the swap counter, After this while loop these off springs $C^{(t+1)}$ and $D^{(t+1)}$, are returned by the function
6. In this way we are generating the off springs from parents using this algorithm.

Mutation:

After returning the off springs by crossover function these off springs are mutated, the mutation is as follows:

1. At first, we will have no of indices to be mutated (MS), this value is randomly generated by taking random number between 1 and 10.
2. After we are generating the MS number of indices randomly, the genes in in these indices of the off springs are going to be mutated.
3. Now we are adding certain amount of noise for these genes (Which are corresponding to these indices) of every offspring.
4. Let's say the gene has a value G, then the noise to be added for the gene is taken randomly from the uniform distribution between -k%ofG and k%of G
5. The MS, and have to be mutated indices are generated for each off spring, like looping through each off spring, and calculating these.

6. So, for an offspring the genes corresponded to the indices generated in this loop are going to add by the noise which is mentioned in 4^{th} point.
7. And also, we observed that If a gene becomes 0 then we are not able mutate that as because all are in multiplication of that, for that we take three random indices and check whether the gene in that index is 0 or not, if 0 then we are assigning a random number which is taken from normal distribution with mean 0 and standard deviation 0.33(this has been fixed after running many generations).this is in each off spring loop.
8. The k is value is varied between 2-5, after observing many generations and the errors we fixed that to 5.
9. And in each gene in offspring checking that every gene value is between -10 and 10 when these are mutating.
10. All off springs after going through these mutations their fitness values will be calculated.

Hyperparameters:

1. **numParameters:** It is the size of the vector i.e., 11, used when the size of the vector is used.
2. **population:** This is the Population size of the generation, we got best results when we take population size as 15.
3. **numParentsMating:** This is the number of parents that are taking to mating, I.e., these parents will undergo crossover, by selecting the pairs by roulette algorithm.
4. **num_generations:** It is the number of generations to run the code. We kept varying by seeing how many requests left.
5. **aa:** it is the weightage given to the sum of errors as mentioned above, we varied this value based on the errors we got by submitting best one, frequent values we used are 3, 2,0,0.01,0.002 etc.
6. **bb:** it is the weightage given to the absolute difference of errors as mentioned above, we varied this value based on the errors we got by submitting best one, frequent values we used are 7,8,100,1000 etc.

Statistical Info:

1. **Number of iterations to Converge:** When we run the code, at first it is converging in very less no of iterations, like 10 to 15 iterations, we thought that the initial population is not diverge, so we changed our fitness

functions and other parameters, then it started converging after 70 to 80 generations.

2. **Errors:** By concerning the difference between the train and validation error, the best error that our algorithm produced is train error: 578612503049.2654 and validation error: 583652508523.2582. This vector performed reasonably well on test data too (based upon the rank in the leader board). Hence, we deduct that the vectors with errors with in this range might be suitable fit for the data.
3. **Range of errors**: From our observation, we reckoned that vectors with errors less than 35*(10**10) overfit the training data and may not perform well on the test and validation data. We also observed that when we keep on decreasing the difference between the validation and test error, the model is getting overfitted on the validation data, which in turn resulted in poor performance on test data.

Heuristics:

1. Initially we are randomly selecting the pairs of parents for crossover by randomly generating two indices from 0 to 5, as we have population as 15, we are taking the top 5 best vectors and selecting them as pairs and doing crossover, but the results are not good, then we studied a research paper which says that roulette algorithm will improve the quality of the selection by assigning reasonable probabilities to the parents based upon their fitness values. For calculating probabilities from fitness values, we have divided fitness value of best 5 fit parents. The selection of parents for mating is based upon their probabilities. We have calculated the probabilities for only the best fit 5 parents. From the best fit 5 parents, we have generated 5 pairs (a parent can be in any number of pairs) which in turn produces 10 children, two children for each pair.
2. In the beginning, we have used point cross-over as it is explained in class. It works fine but we thought that the point cross over may not induce enough uniqueness or diversity in the children. So, we looked up many research papers and decided to use “Highly Disruptive Crossover- Heuristic Uniform Crossover (HUX)”. We have chosen this cross over heuristic because of the fact that it changes the child vector quite considerably from the parents and can u helpful by imparting diversity and uniqueness to child and also retaining the good traits of the parents.

3. At the beginning of the algorithm, we are generating required number of copies of overfit vector and mutating them to generate the initial population. For this mutation, we have used random.uniform to generate the number with in a range depending upon the values. But we observed that the algorithm is converging very quickly within 4-8 generations. We found that the lack of diversity in the initial population is making the algorithm to converge quickly. So, we observed some blogs and articles. From those, we got to know that instead of doing uniform distribution usage of normal distribution can be help in imparting diversity to the initial population. After running many observations and considering the number of generations taken to converge and also the range of values in overfit vector, we have decided to keep the value of standard deviation for the normal distribution as 0.33-x, where x is value of the gene and it is also the mean for the distribution.
4. For carrying out mutations in every generations, we have random.uniform to generate number uniformly within the range of (-0.05*x, 0.05*x), where x is the value of the gene. We have observed when the value of the gene is considerably low when the mutations are much effective in changing the value. So, we have decided to use random.normal during mutation, if the value of the gene is considerably low.
5. Based upon the performance of the vector on train, test, and validation data, we keep on changing the values of hyper-parameters to obtain better vectors. Sometimes, we have also taken range of errors into consideration for deciding the hyper-parameters.

BEST ERRORS ACHIEVED:

Train error: 578612503049.2654

Validation error: 583652508523.2582.

By our conceptions and parameters, these are the best errors that we have achieved. We observed that many vectors converged around this range of errors.

REASON FOR CHOOSING THE BEST FIT VECTORS AS THE VECTORS FOR OVERALL BEST PERFORMANCE

We are considering both the sum and absolute difference of train and validation errors for calculating the fitness of the vectors.

i. Reason for choosing sum in calculating fitness: If the sum of errors of the vectors is low, then it means the model is good at mapping to the dataset.
ii. Reason for choosing difference in calculating fitness: The maintaining the difference between the validation and train error low, we can be sure that the model is performing equally on both train and validation data sets.

The linear combination of sum and absolute difference of the train and validation errors helps to prepare the model so that it won't overfit on dataset (since we are trying to maintain low absolute difference between the train and validation data sets, which means considerably equal performance on both train and validation data sets) and also make sure that the model doesn't underfit (since we are trying to maintain the sum of errors low, the model fits data well). Hence considering the linear combination of sum and absolute difference of the train and validation errors make the model not to either overfit or underfit the datasets. This makes the model not to exhibit any kind of bias towards the training set and also makes sure the model picks just enough information to predict the underlying patters of data. This is typically the behavior of an ideal model.

Hence, we believe that top 10 vectors perform well on the unseen data.