MDL PROJECT REPORT

Team Number: 50
Team Name: LearningMachinesFromData
Team Members:
T.H.Arjun (Roll number: 2019111012),
Gokul Vamsi Thota (Roll number: 2019111009)

Summary of Genetic Algorithm:

The initial overfit vector is used to generate a population of required size, for the genetic algorithm to work. This generation of initial population was achieved by applying a normal distribution over this vector, and by randomly introducing some noise in a few positions. We begin calculation with this initial population.

1) We calculated fitness values for all these chromosomes in this population, with the help of a fitness function. Then, we separated this population into two categories - mate pool (say parents), and rest (say children).

2) We now choose a fixed number of fittest parents to be in the mate pool, and also, choose a fixed number of chromosomes in the rest of the group too (children), to be a part of the mate pool. This is to ensure that there is scope for new generations to progress further.

3) After choosing the fit guys from both categories as above, there are still some positions left in the mate pool, these can be filled up by the other fittest chromosomes in the current population we didn't choose yet. Note that all the selections above have been done in a 'greedy' way (Chromosome with high fitness is prioritized first).

4) Now, on the chromosomes present in the mate pool, we choose a fixed number of pairs to mate (randomness is involved such that chromosomes with higher fitness values have more priority of being chosen), such that each pair generates two offsprings, possessing traits of the two based on the crossover function. The breeding is constrained such that, exactly (POPULATION_SIZE - MATE_POOL_SIZE) number of offsprings are generated.

5) Call the currently generated set of offsprings as new_children. Now, the set of parents is updated such that, we consider only the 'MATE_POOL_SIZE' fittest

number of chromosomes (in a greedy way), from the population this generation has begun with (excluding new_children).

6) We update the children set now, by considering all the chromosomes which have just obtained (new_children). This way, in the set of parents, we have all the fittest chromosomes till current generation (excluding offsprings generated in this generation), and in the set of children we have all the offsprings generated in the breeding that took place in this generation. Combining both these sets, we have a (POPULATION_SIZE) number of chromosomes, which will act as a population for the next generation.

7) Now that we have the population for the next generation (in the sets of parents and children), we can continue with the same algorithm from step 2, and obtain populations of as many number of generations as we want.

Fitness Function:

The fitness functions we used were modified over different days and across different generations. The fitness function was calculated in the following way:

1) We included parameters called FACTOR and SUM_FACTOR. The parameter FACTOR described the weightage that should be given to the validation error relative to the train error obtained, and the parameter SUM_FACTOR described the weightage that should be given to the absolute difference between these two errors relative to the sum of these two errors. These parameters were varied to obtain desirable populations.

2) Based on these parameters, we computed some values as explained below:

a = ((TRAIN_ERROR + VALIDATION_ERROR * FACTOR))
b = ((ABSOLUTE_DIFFERENCE_OF_ERRORS * SUM_FACTOR))
e = a + b

3) We now used a constant value, K = 1e15, and computed the fitness value of a chromosome to be (K / e), as fitness is inversely proportional to errors. The constant was introduced just to obtain meaningful values for fitness.

4) The chromosome with higher fitness value is considered more desirable to be passed on to further generations.

Crossover Function:

The crossover function takes two parents as inputs and generates two offsprings (based on traits of the parents) and returns them as output. We are taking help of the blend crossover algorithm (BLX-a-b). It computes the offsprings in the following way:

1) Pass the two parent arguments (say X and Y), in such a way that fitness of X is greater than fitness of Y.

2) Consider the absolute difference of these two parents (absolute difference vector) to be d. Consider two parameters a and b, which are both some random real values between 0 and 1.

3) Now, compute the two children in the same way as described below:
For each of the n positions, let's say that the value of the component in that position of the parent X is $p1_i$ and the value of that position's component in parent Y is $p2_i$, and let's say that d_i is in i^{th} position of d. and let's say that the value of the component in that position for the child would be c_i
if $p1_i <= p2_i$:
c_i = Some random real number between ($p1_i$ - ($a*d_i$)) and ($p2_i$ + ($b*d_i$))
else:
c_i = Some random real number between ($p2_i$ - ($b*d_i$)) and ($p2_i$ + ($a*d_i$))

Using this algorithm, we compute all the n positions of both the child vectors, based on the parent vectors.

Mutation Function:

This function takes a set of children as an argument and returns the set of children obtained after applying mutations, as explained in the following way.

1) Define a parameter (say m), which represents the percentage of the current component that is used in calculating noise via randomization (it is a real number between 0 and 1, 1 represents 100%). Define another parameter, (say MUTATION_SIZE), which decides the number of positions of a child chromosome that are going to be altered with a particular amount of noise. These parameters were varied across generations, for obtaining desirable populations.

2) For every child vector with i^{th} position $child_i$, generate a noise vector, such that the i^{th} component in the noise vector is given by the step:
$noise_i$ = Some random real number between $(-m*child_i)$ and $(m*child_i)$.

3) After computing the noise vector, choose MUTATION_SIZE number of positions in that child vector randomly and add the value of $noise_j$ to the j^{th} position chosen, and leave the other positions of child vector unaltered, because we don't want mutation to alter the entirety of the chromosome.

4) In this way, we obtain all child vectors after applying mutations. We typically considered the value of m to be 0.05 and MUTATION_SIZE to be 5 in most cases (as there are a total of 11 positions).

5) Ensure that all the components of all the mutated child chromosomes are in the valid range (between -10 and 10).

Hyperparameters:

1) MUTATION_SIZE : It is the number of places that will be mutated randomly in th 11 D Vector. We varied this between 3 and 5. When we thought that our algo was not moving forward and converging we increased it and when all vectors were coming bad we reduced it. The sweet spot is around 3-5.
2) POPULATION_SIZE: This is the population size for each Generation. It is best around 30.
3) SELECT_TOP_PARENTS: It is the number of top parents in a population that will be selected for sure when selecting for matepool. At the start of the algorithm we kept this low as we wanted more children in the mate pool as parents were bad vectors with high errors. Also we wanted a lot of variety to keep the algorithm from converging which will be given by the child vectors and mutations. Later when everything was converging we increased it so that we perfect the perfect.
4) SELECT_TOP_KIDS: It is the number of top parents in a population that will be selected for sure when selecting for matepool. Initially we kept it high so that we get variety from the children and since we didn't want a minima early on and wanted to search the whole search space. Later when the algo proceeds we reduce it so that we don't get a lot of variety.
5) MATE_POOL_SIZE: This is the total number of individuals selected for mating in the russian roulette. We kept it around 10 initially since we wanted a lot of children from parents, for variety and to stop the algorithm from reaching

minima very early on. Later we increased it to 15-20 based upon the vectors produced by the algorithm.

6) MAX_GEN: It is the number of generations to run the code for. We kept varying it on the number of requests that we had left.
7) FACTOR: As mentioned above it is the weight given to Validation error. We varied it between 1 and 2.
8) SUM_FACTOR: It is the weight given to the absolute difference as mentioned above. We varied it between 0 and 3.

Statistical Info:

1) Number of Runs to Converge: On our testing this varied a lot and depended on the random functions of numpy that we were using. In one of our runs it converged after 100 runs and in some it converged earlier or later also.
2) The best value our algo produced was 52053036962.50848 for train error and 99278182014.10391 which to us seems really good numbers but this seem to perform very badly on the test data, which might be because it is overfit on both sets now, which is a disadvantage in genetic algorithms.

Heuristics:

1) BLX a-b crossover: The first breakthrough we did was implementing Blend CrossOver (BLX a-b). Initially we had a single point crossover and then a multipoint one. BLX a-b crossover was performing the best compared to the other two in case of decimal values, while the other two were optimised for binary.
2) Initially we were selecting from matepool in a circular pattern. On reading research papers we came to know that Russian Roulette is the best way to do this based on probabilities. We were getting better results and better variety. Russian roulette method is employed in achieving selective breeding among different pairs of chromosomes, by prioritizing parents with higher fitness values. Based on the fitness values of all the chromosomes in the mate pool, this set of fitness values were normalized and further each fitness value was divided by the sum of fitness values of all chromosomes. This provided us with a probability vector, such that the sum of all probabilities is 1, and the fitter chromosome would clearly be having a higher probability value. And

thus, now two chromosomes are randomly chosen, with their probability as computed in the probability vector above. This randomization has been effective in the long run.

3) Selecting some top parents for sure and Selecting some top children for sure was one another heuristic we applied. It helped us to make sure that we get genes from both these groups and that our algorithm keeps on getting new chromosomes.
4) We tried many ways to get the initial population during the course of the project. First we tried taking the overfit vector and randomly randomly mutating it with numbers between -10,10 but this was not showing good results. The search space was too wide. So we decided that we will scale the overfit vector by multiplying by a random number between 0 and 1.2. This also didn't work well. The algorithm converged too quickly. Then we understood that we should use Normal Distribution. This is because we want the values around the initial value of overfit but distributed widely but with lower probabilities. So we choose a normal distribution with mean of gene value and standard deviation as 0.33-x for mutation of overfit. This was giving us good results. Then when this method converged, we noticed that the all-zero vector was performing well. So we decided to mutate the all zero vector with values from overfit and try. But this did not yield results. But what yielded the result was using normal distribution on All-zero vector at random places.
5) Other Heuristics we applied was on mutating the children. Initially we mutated with random numbers. That didn't work well. Then we scaled them by a factor between 0 and 1.2 that didn't work well too. Finally what worked was adding a percentage of value to the gene. We realised that what should be added or subtracted should be a percentage of the current value. So we will add/subtract bigger values for big numbers and the same goes for small numbers. We kept varying the mutation between +-5% to +-15% of current value but the sweet spot is at +-5%.
6) Another heuristics we applied was varying the fitness function as mentioned above with parameters to get desired population.

Best Train and Validation Error Observed:

In the due course of requests made for this project, we observed that, the best set of values we obtained for the train error and validation error are:

Train error: 52053036962.50848,
Validation error: 99278182014.10391

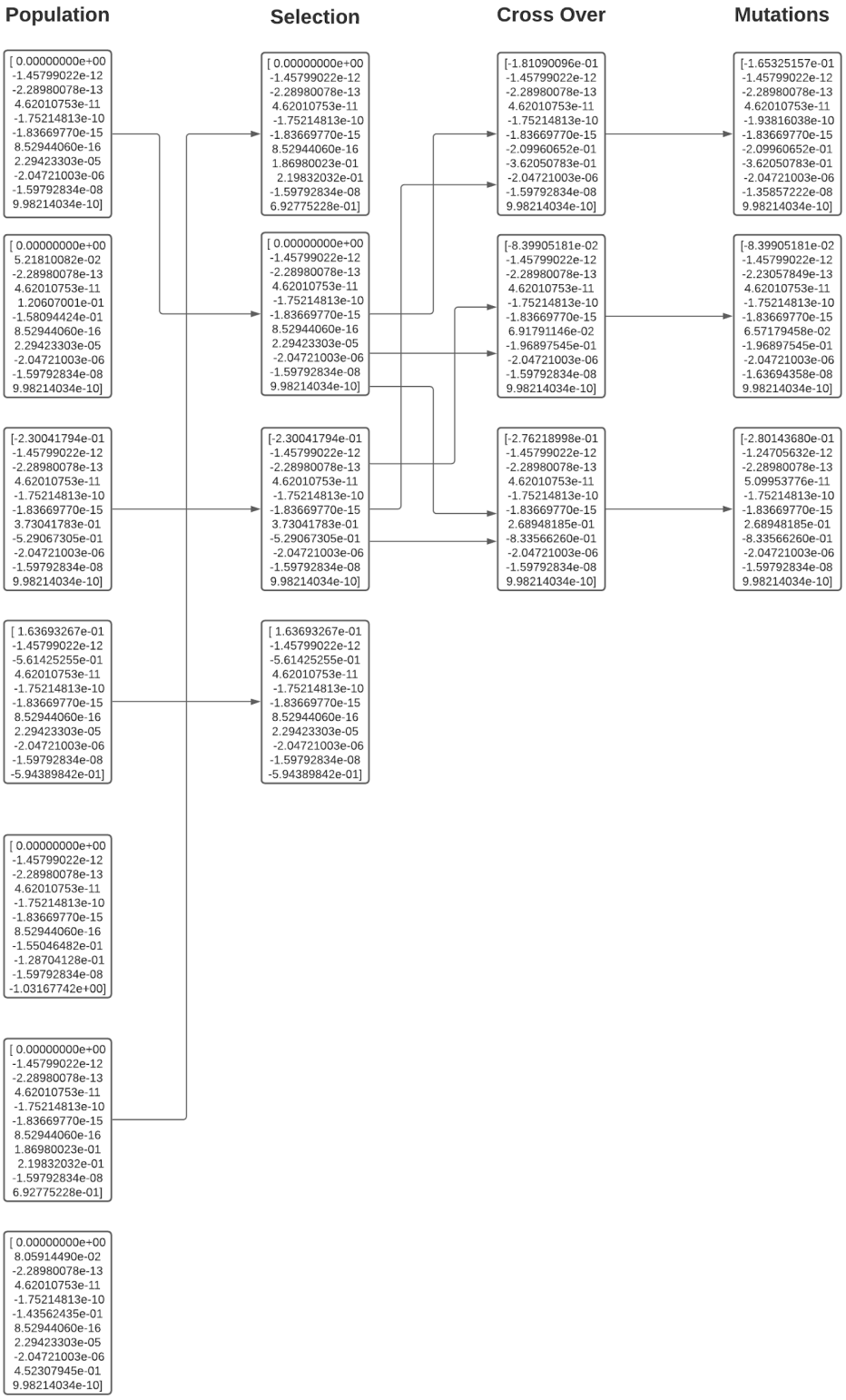
These values were obtained after running the algorithm for many generations and convergence was clearly observed.

How we chose our Top 10 and reason for overall optimal performance:

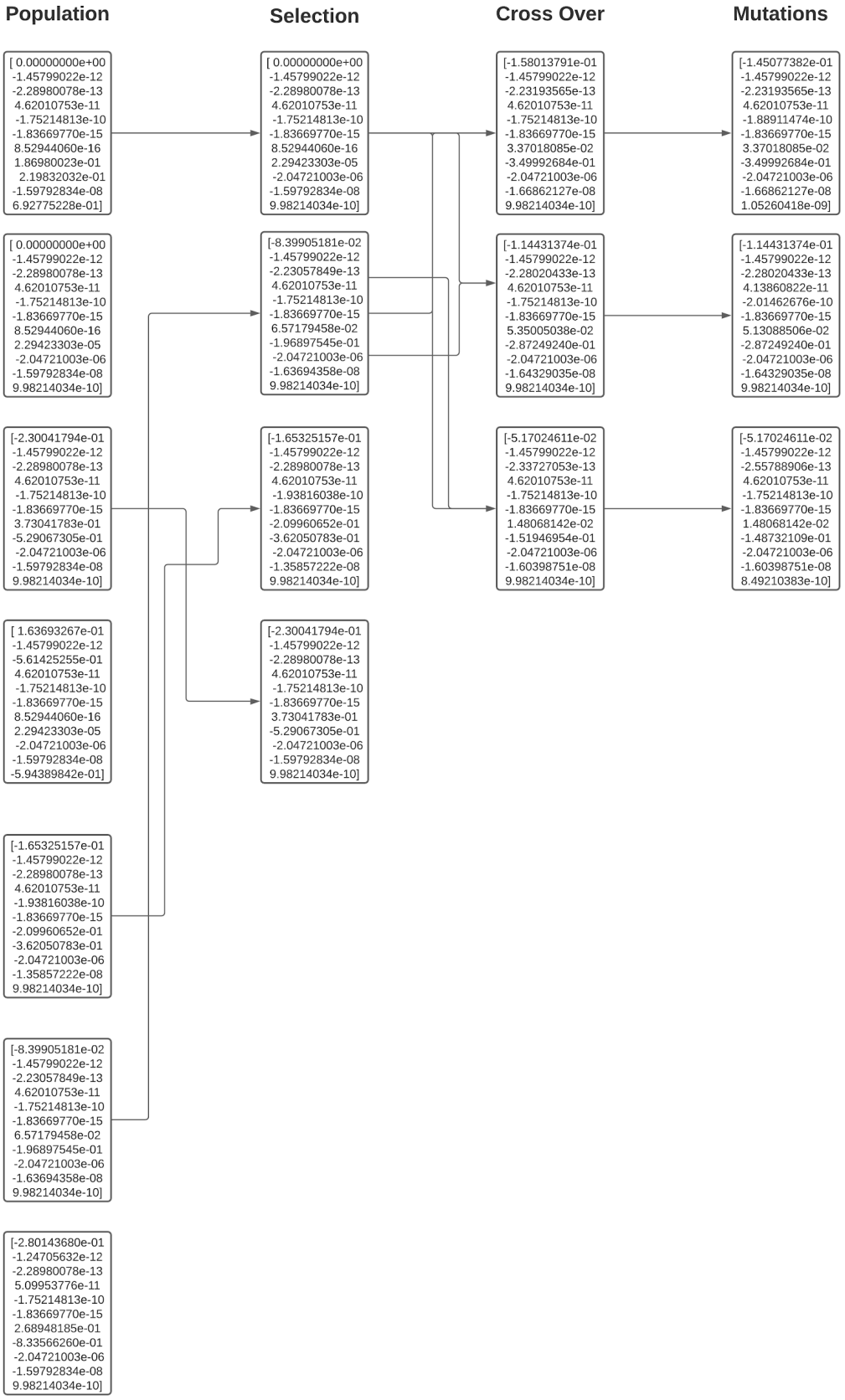
We chose our top 10 based upon the sum of errors that we got and the difference in error. If the sum is low that means that our vectors map to the dataset very well (resulting in low errors) , we also took vectors with low difference in errors as this means that they represent both datasets equally. Thus performing well on both. This is typically the expected behaviour of an ideal model, because it doesn't exhibit any sort of bias towards the training set (like the overfit vector provided) while simultaneously learning just enough information from the data without being underfit (as errors are being minimized), as it produces similar/expected results in such scenarios. This according to us is the most optimal way of choosing our top 10 vectors and we believe they will perform the best in unseen data.

Diagrams for the Algorithm for 3 Generations

Generation 1 : Initial population from Overfit + Normalised Mutation



Generation 2 :



Generation 3: