MDL Project - Bad Juju

Aakash Aanegola (2019101009)
Aakash Jain (2019101028)

Genetic Algorithms

Genetic algorithms work on the simple premise of imitating evolution. To implement the genetic algorithm required for this project, we used an object-oriented approach where each individual (weight vector, fitness, etc.) in the population was represented by an object. To make our code modular and extensible we abstracted all the intricacies of the algorithm, and only provided functions to undertake the roles of the algorithm.

```
class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = None


    # Calculate the fitness of the individual (lexicographic distance)
    def set_fitness(self):
        errors = get_errors(ID, self.chromosome)
        self.fitness = WEIGHT * errors[1] + (1-WEIGHT)*errors[0]

    # Mutate to change some genes
    def mutate(self):
        new = []

        for i in range(11):
            if random.uniform(0, 1) < MUTATION_PROBABILITY:
                new.append(self.chromosome[i] + np.random.uniform(-0.1, 0.1)*scale[i])
            else:
                new.append(self.chromosome[i])
            if new[-1] > 10:
                new[-1] = 10
            if new[-1] < -10:
                new[-1] = -10
        self.chromosome = new
```

The individual class (above) contains the chromosome (in this case the weight vector) and its fitness score (that was defined as some linear combination of the training and validation errors). To set the fitness (with the API call) and to mutate the choromosomes, we created functions.

The parameters of the genetic algorithm were changed extensively and are explained below:

```
# The number of individuals in the population
POPULATION_SIZE = 10
# The probability of a gene mutating
MUTATION_PROBABILITY = 0.1
# The percentage of the population that is carried over into the next population
ELITE_PERCENTAGE = 0.3
# The percentage of the population that is bred to form the next generation
BREED_PERCENTAGE = 0.7
# The number of generations
GENERATIONS = 81
# The linear combination fraction of the validation error
WEIGHT = 0.5
```

From a very high level perspective, our algorithm works in the following way:

- Generate initial population (if we already have a starter population, simply read it from a file) and set their fitness scores. Generating the population was done randomly for the first sample using the scales from the overfit vector.
- Sort the population based on their fitness scores.
- Breed the individuals selected for breeding (defined by ELITE_PERCENTAGE) and generate 7 children randomly.
- Create the new generation with the selected individuals from the old population and the newly bred children.
- Calculate the fitness scores of the new population, and repeat from step 2.
- **EXIT:** Once the API calls have been exhausted, dump the latest population for the next iteration.

Fitness Function

The fitness function we used (final result) is a linear combination of the training and validation errors.

$$f(val, train) = (1 - w) \cdot train + w \cdot val$$

As we can infer from the function, the lower val and train are, the lower the fitness score will be. This means that a low fitness score is desirable (the inverse could've been taken, but sorting based on this value in ascending order was viable and hence we chose not to). The best individual is the one with the lowest fitness score. We varied w everyday depending on which error was higher (val/train) in an attempt to balance them.

Crossover Function

Our crossover function treats each gene in the chromosome (weight in the weight vector) independently. Although most crossover functions choose an arbitrary crossover location and select the left part from one parent and the right part from another parent, we decided not to do it this way because there is no dependence of a weight on its neighbours. Each weight has a 50% chance of being taken from either of it's parent and hence we reconstruct a new chromosome randomly selecting genes from either of the parent chromosomes.

Mutations

The mutations were applied to the entire population after the selection and crossover stages. Each gene in a chromosome had a chance to mutate. The mutations were anywhere in the range

$$[-0.1, 0.1] \cdot scale[i]$$

where scale[i] represents the scale (order) of the ith gene. The mutation value was then added to the current gene value.

```
# The scale vector
scale = [0, 1e-12, 1e-13, 1e-11, 1e-10, 1e-15, 1e-15, 1e-5, 1e-6, 1e-8, 1e-9]
```

Hyperparameters

```
# The number of individuals in the population
POPULATION_SIZE = 10
# The probability of a gene mutating
MUTATION_PROBABILITY = 0.1
# The percentage of the population that is carried over into the next population
ELITE_PERCENTAGE = 0.3
# The percentage of the population that is bred to form the next generation
BREED_PERCENTAGE = 0.7
# The number of generations
GENERATIONS = 81
# The linear combination fraction of the validation error
WEIGHT = 0.5
```

These are the parameters used for the final generation, but changed extensively over the course of the project.

- **Population size:** The population size was set to 10 from the very beginning as it gave us a good balance between number of generations and population size. The final submission is also 10 vectors and also the number of API calls was always divisible by 10.
- **Mutation probability:** The mutation probability was set to 0.1 percent (very arbitrary choice) after trying values between 0.05 and 0.4.
- **Elite and Breed percentages:** These values were also arbitrary after trial and error with values between 0.1 and 0.4 for elite percentage and 0.5 and 0.9 for breed percentage. The low value of elite percentage ensures that we allow new individuals to enter the population, and the high value of breed percentage ensures that we have diversity in our population.
- **Generations:** The number of generations is simply the number of available API calls/the population size.
- **Weight:** The weight of 0.5 ensures equal contribution of training and validation errors to the fitness function.

Statistical Information

- **Generations to converge:**
  The genetic algorithm will keep trying to get better results, but we didn't let it run till that point. If we made the training and validation errors too low, the resulting model would overfit the larger set training+validation. However assuming that the lowest error is around 1e10 (overfit training error should be a lower bound) for both training and validation, and assuming an exponential decay (it gets exponentially harder as time goes by to reduce the error) we estimate that it would take > 20000 generations to converge to the lowest point (global minima). However, the algorithm does find local minima fairly quickly (50-100 generations) and then waits for a favorable mutation to make a jump to the neighborhood of another local minima.

Heuristics

- **Finding individual optimum weights:**
  A method we tried to get a good starting point was to find optimal values for each of the weights and then combine them. This didn't work well as locally optimising with some constant values of the other weights will not give us the true optimum value.
- **Taking extremes and weighted average:**
  Another method we tried was to take the extreme values for each point (-10, 10) and then take a weighted average of the errors we obtained from them.

$$w_i = -10 + \frac{E_i(10)}{E_i(-10) + E_i(10)} \cdot 20$$

- **Taking extremes and weighted average of square roots:**
  Since we are optimising on mean squared error, we thought we should attempt the weighted average method but this time taking the square root of each of the errors.

$$w_i = -10 + \frac{\sqrt{E_i(10)}}{\sqrt{E_i(-10)} + \sqrt{E_i(10)}} \cdot 20$$

- **Using the scale of the overfit vector and generating random points:**
  We used the order of the weights of the overfit vector, generated random numbers between -10 and 10 and multiplied them with the scale to get our population. This gave us a good initial population to start with, and we mutating using the same method, but the mutations lie between -0.1 and 0.1 times the scale.
- **Overfitting on validation:**
  Since the given vector overfits on the training data, we thought we could overfit on the validation data to balance it. However we didn't stop it soon enough and obtained a vector that was overfit on the validation data.

Optimum train and val

Our algorithm did generate vectors with $train = 2 * 10^{10}$ and $val = 9 * 10^{10}$ on independent vectors, but together we chose to stop the algorithm when $5 * 10^{10} \leq train \leq 1.2 * 10^{11}$ and

$9 * 10^{10} \leq val \leq 1.5 * 10^{11}$. Since we are aware that the model overfits when $train = 1.1 * 10^{10}$, it would be optimum to try and make $train = val$ and make sure they're in the range of $\approx 10^{11}$, and hence this is where we decided to stop the algorithm. We believe that these weight vectors will perform well on unseen data as they do not 'overfit' the training or validation data and still are comparable to the overfit models error values.

Diagrams

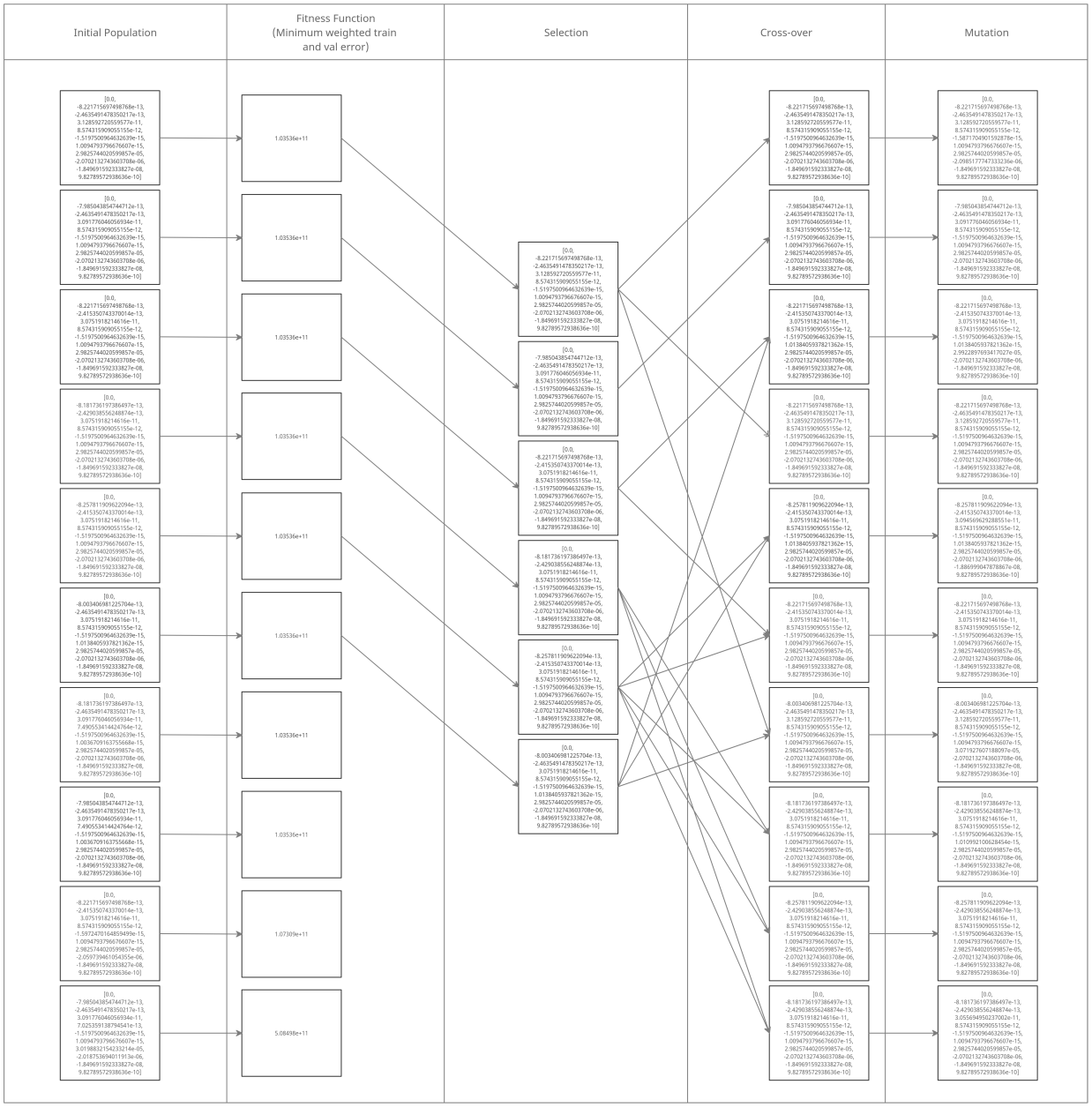

Fig 1: Genetic algorithm iteration for arbitrary initial population

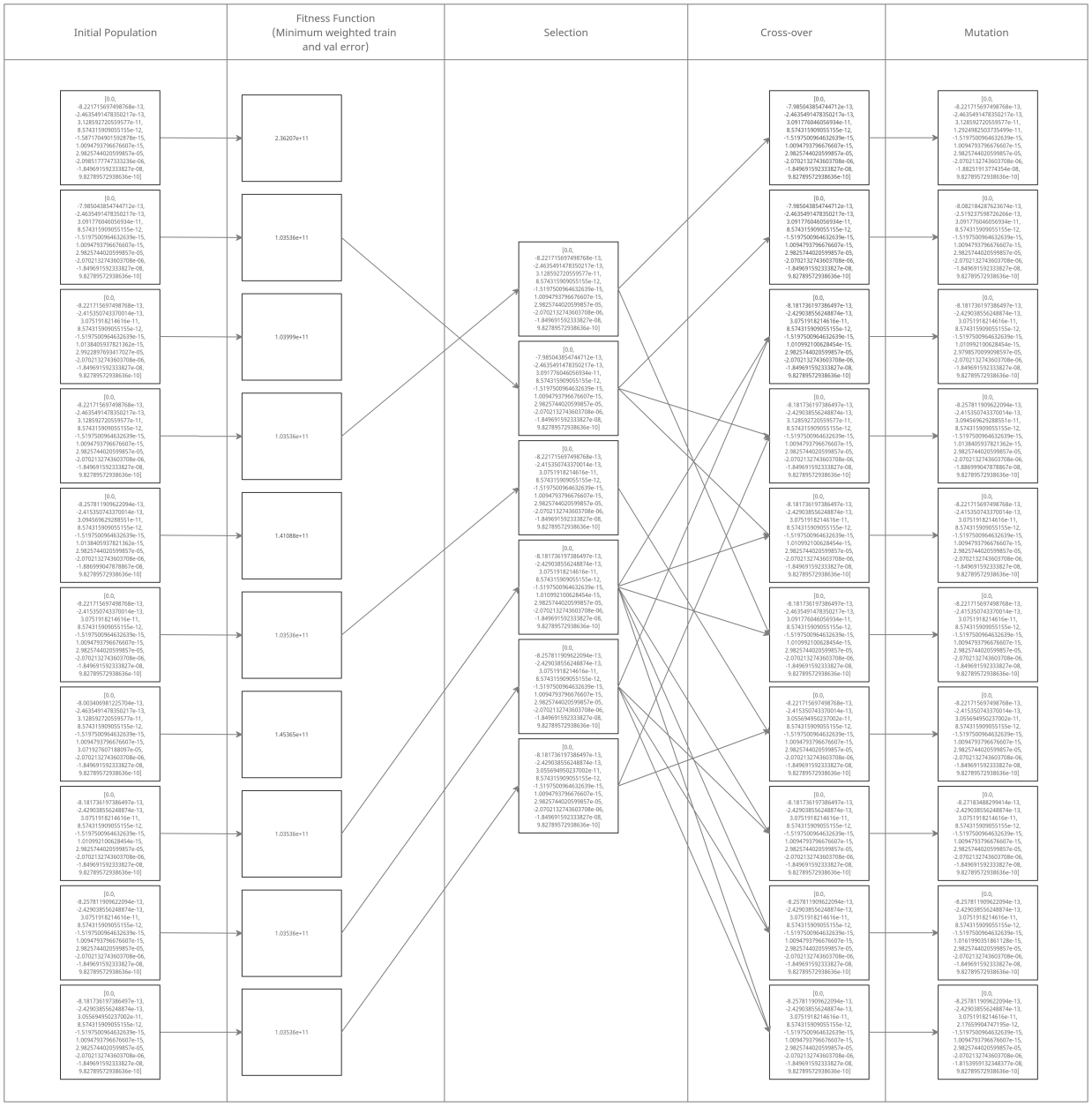

Fig 2: Genetic algorithm iteration with mutated previous iteration population

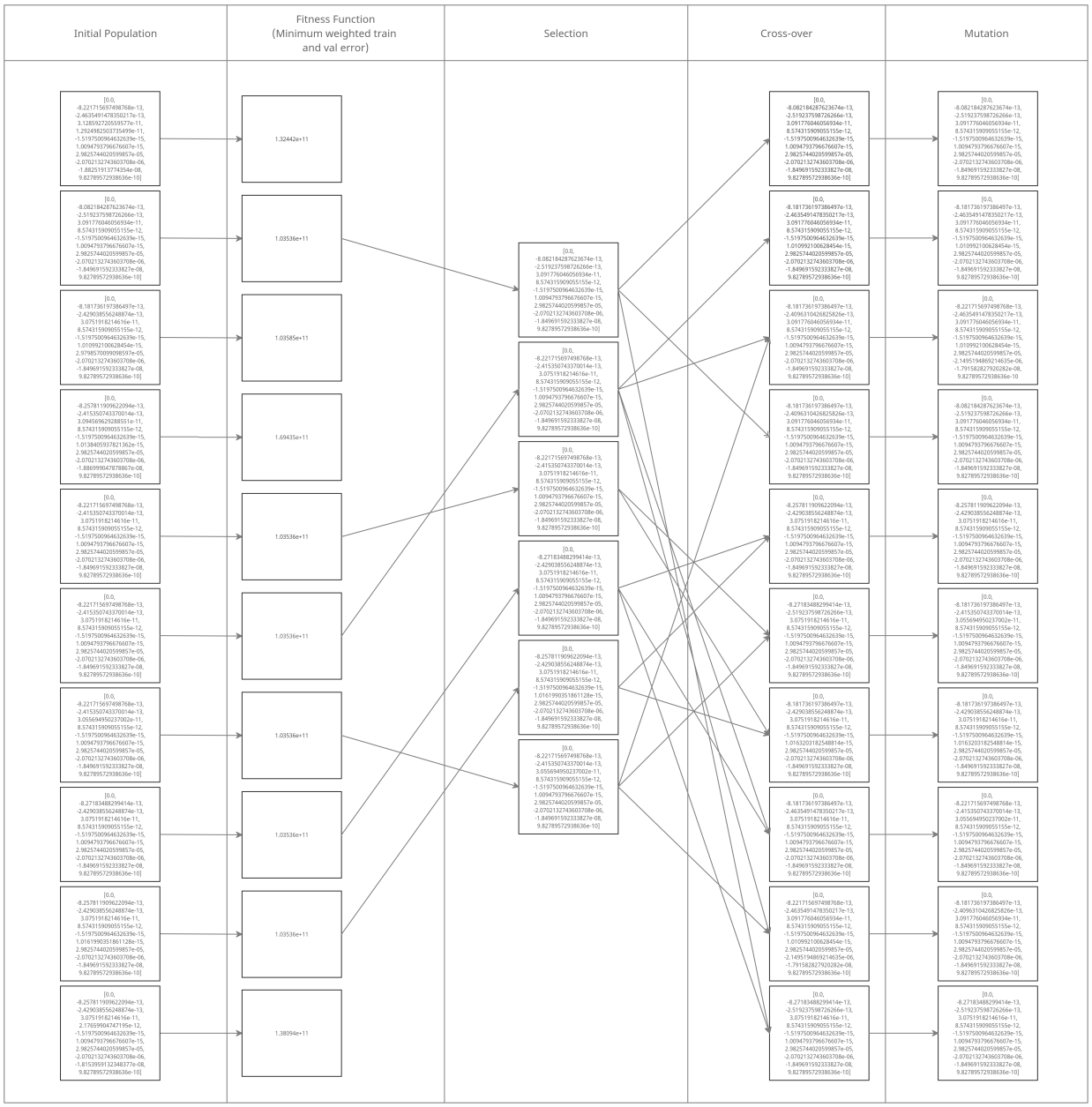

Fig 3: Genetic algorithm iteration with mutated previous iteration population