MDL PROJECT REPORT

Team no. 107
Prateek Sancheti, 2019111041
Pranoy. J, 2019115004

Genetic Algorithms - A brief introduction

A genetic algorithm uses a population of possible solutions that function as the initial generation and then applies natural selection and mutation to this generation to get the next generations containing solutions that are more optimal to the problem. Generally, a fitness selection is used to simulate natural selection and a crossover is used to generate the next generation from its parents.

A genetic algorithm generally consists of the following elements:

- A genetic representation of a solution
- A function to generate new solutions
- A fitness function to implement natural selection
- A selection function to select parents who move on to the next generation
- A crossover function to generate a new generation
- A mutation function to mutate the current generation.

The Genetic Algorithm Implemented

- *The genetic representation of a solution*:

As per the question, the solution required would be a vector, and hence the genetic representation of the solution will also be in the form of a vector.

- *A function to generate new solutions*:

The initial generation had already been provided along with the project details. So, these values were adopted into the program.

```
def initialize_population():  # load previous population as per status
    np.savetxt( './temp.txt', np.loadtxt( "./saved_populations.txt", delimiter=',' ), delimiter=',' )
    return np.loadtxt( "./saved_populations.txt", delimiter=',' )
```

This function would load the previous generation and the first load would occur from the temp.txt file provided.

- *A fitness function to implement natural selection*:

This function would snd a query to the server along with a vector to receive the fitness value.

```
def cal_pop_fitness(vector):
    # Calculating the fitness value of each solution in the current population.
    for i in range(40):
        vector[i][0]=0
        temp = list( vector[i][1:] )
        err = client.get_errors( secret_key, temp )
        total_error = err[0]+err[1]
        vector[i][0] += total_error
        print( "query number:- ", i, err )
    return vector
```

This function returns the error created by a certain vector and the fitness of the vector is determined by applying the following rule: *lower the total error, higher the fitness*.

- *A selection function to select parents who move on to the next generation*:

To avoid losing an optimal solution while transitioning between generations, the best vector from each generation is carried on to the next.

```
    parents = vector[:20, 1:]
```

- *A crossover function to create the next generation*:

To obtain a new generation from the previous one, two parents are selected at random, a certain point (crossover point) is decided randomly and the vectors chosen are split into two at the crossover

points and the latter halves are interchanged. This is a single-point crossover function.

```
def crossover(parents):
    offspring = np.empty( (20, 11) )

    crossover_point = random.randrange( 1, 9 )

    for k in range( 20 ):
        # Index of the first parent to mate.
        parent1_idx = k
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:crossover_point]
        # Index of the second parent to mate.
        if(parent1_idx!=19):
            parent2_idx = parent1_idx+1
        else:
            parent2_idx=0
        offspring[k, crossover_point:] = parents[parent2_idx, crossover_point:]
    return offspring
```

- *A mutation function to mutate the current generation*:

This has been implemented in the form of 2 functions, the first (*mutation*) which calls the other and provides the vector along with the other variables and appends the new mutated vector back into the generation, and the second (*mutate*) which performs the mutation on a vector with a fixed probability.

```
def mutate(val, start, stop,lim):
    temp = val * random.uniform( start, stop )
    if (temp < lim*-1):
        return mutate( val, -10 - val, stop,lim)
    if (temp > lim):
        return mutate( val, start, 10 - val,lim)
    return temp


def mutation(offspring, parents):
    for idx in range( 20 ):
        for g in range( 0, 11):
            offspring[idx][g] = mutate( offspring[idx][g], -2.0, 2.0,10)

    for idx in range(20):
        for g in range( 0, 11 ):
            parents[idx][g] = mutate( parents[idx][g], -2.0, 2.0,10 )
    return np.append( parents, offspring, axis=0 )
```

The Best vector Obtained

The best vector we obtained using the code is as follows:

[-0.0, 3.041197317142537e-19, 1.872054489906164e-09, -1.9889022547238325e-08, -0.1925159618028633, 2.33487630768748e-19, 2.851237000064995e-22, -1.5927927797148508e-13, -4.126167819980085e-13, -2.286558923393001e-15, -7.944189639969726e-13]

The error obtained in the case of this vector was:

3191814440386.9736

Hyperparameters

The hyperparameters used in this code were:

- Pop_num = 40, the number of vectors in a generation
- Instead of declaring a clear number that determines the probability of mutation, a function (*mutation*) has been implemented such that mutation may or may not occur to a vector.

```
def mutate(val, start, stop,lim):
    temp = val * random.uniform( start, stop )
    if (temp < lim*-1):
        return mutate( val, -10 - val, stop,lim)
    if (temp > lim):
        return mutate( val, start, 10 - val,lim)
    return temp


def mutation(offspring, parents):
    for idx in range( 20 ):
        for g in range( 0, 11):
            offspring[idx][g] = mutate( offspring[idx][g], -2.0, 2.0,10)

    for idx in range(20):
        for g in range( 0, 11 ):
            parents[idx][g] = mutate( parents[idx][g], -2.0, 2.0,10 )
    return np.append( parents, offspring, axis=0 )
```

- The crossover point is randomized between indices 1 to 9 of a vector.

Heuristics

1. Tried to implement a fixed probability for mutation: Initially, a fixed probability for mutation was attempted, but the method currently implemented using recursion provided better error values. Therefore, the switch was made.
2. Adding weights to errors: Initially, it was agreed upon to add a weight to err[1ns were not random but a result of the training process. As su] but this was not done because towards the final stretches, the population, both training and validation errors would be equally important in these cases.

Statistical Information

Total error at the end of generation 1: 54301305716l0.094
No. of generations: 120
Total error at the end of generation 120: 3191814440386.9736
No. of vectors per generation: 40