AI Coursework Report

Task 1 - Classifier:

The classification error procedure is as follows:

- We go through every point in either the training set or the validation set
- We then classify the point using the K-Nearest Classifier by calculating the Euclidean distance to all the points in the training data set, then sorting the distance in ascending order and picking the first K distances
- Then we determine the label of a point by seeing the label with the highest count in the points of the K-selected distances
- Finally, once we classified all the points in the training set or the validation set we compute the error by using the following formula:
    - number of misclassified points / total number of points
- Euclidean distance formula used is $\sqrt{\sum_{i=1}^{3} (p[i] - q[i])^2}$

K	Training Classification Error	Validation Classification Error
7	0.0278	0.0463
19	0.0250	0.0519
31	0.0333	0.0519

Task 2 - Regression with Genetic Algorithm:

A. Genetic Algorithm Design:

- objective function?
  - Given that we are facing a classification problem and the number of classifications is just 2 I decided to go with the simple and intuitive objective function that simply gives 1 point for classified points in the training set and takes away 1 point for misclassified points
  - One advantage is that this objective function has a very direct mapping to our goal which is minimizing the classification error
  - $$max_{a_0,a_1,a_2} \sum_{(x,y,z)\in P} g(x,y,z) \quad \text{where } g(x,y,z) = \begin{cases} 1 & \text{if } z \geq f(x,y) \text{ and class } 1 \\ 1 & \text{if } z < f(x,y) \text{ and class } -1 \\ -1 & \text{otherwise} \end{cases}$$
- parameters to be encoded in the individuals' chromosomes?
  - simply a chromosome will be represented as a vector of 3 real-values [a0, a1, a2]
  - real-valued encoding is chosen for mainly 3 reasons 1st is that it provides a continuous representation of the solution space, 2nd it is fairly easy to implement with no need to do any conversion to a lower-level encoding, 3rd it is more intuitive to use simply a list of 3 real-values given that those values are directly used to guide the search in the objective and fitness functions
- mapping between genotype and phenotype?
  - There is no mapping between genotype and phenotype both will be the same
  - justification described in the above bullet-point
- initialization of the population?
  - I had 2 thoughts for initialization the population first is defining a fixed value population and having a for loop that generates random values for [a0, a1, a2], but this seemed to be very random
  - The 2nd approach that is used by the genetic algorithm tries to force having a diverse population using combinatorics by forcing having different combinations of points in different regions along the limits of (a0, a1, a2)
  - I am enclosing the pseudocode for population creation as it is much easier to understand the idea from the code

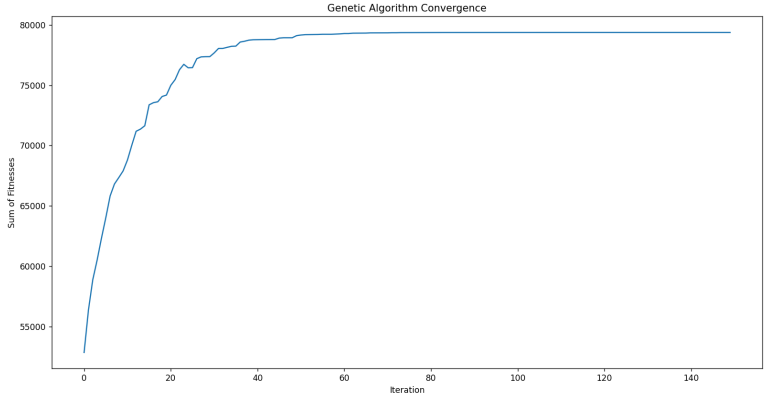
```
# pseudocode for population creation
# random(a, b) produces a floating-point random number
between (a, b) inclusive
factor = 0.5
for (a0 = 0; a0 <= 2; a0 += factor)
    for (a1 = -2; a1 <= 0; a1 += factor)
        for (a2 = -1; a2 <= 1; a2 += factor)
            individual = [random(a0, a0 + factor),
random(a1, a1 + factor), random(a2, a2 + factor)]
```

- population size?
  - after doing a rough time complexity analysis it turns out that the genetic algorithm runs with the following complexity O(nmq)
    - n: population size
    - m: size of training data

        - q: maximum limit of iterations for the genetic algorithm
    - here given that we have m = 360 and q = 150 (q was hard coded as 150), we can now try to figure a value for n
    - given the assumption that 10,000,000 operations can run in roughly 1 second in c++, we would substitute in the equation 10,000,000 = 360*150*n
    - abs(n) = 185, if we have used a factor = 0.4 as our factor in the previous procedure we will have n = (2/0.4)^3 = 125 which seemed sensible but after some experimenting this value seemed to be time-consuming in Python, and after trying a factor = 0.4, where n = (2/0.5)^3 = 64 it seemed to be much faster and yield to some extent very similar values to the population with 125 elements
    - So finally we can say that our population is 64
- the way for assigning fitness values to individuals?
    - simply our fitness function is equal to the objective function of an individual plus an excess value to avoid having negative values in the fitness function
    - fitness(individual) = objective(individual) + len(training_set)
    - we have excess = len(training_set) because the lowest value that the objection function can produce is -len(training_set)
    - fitness(individual) ∈ [0, 2*len(training_set)]
- selection procedure?
    - we are using the Roulette Wheel Selection
    - More fitter individuals have higher probabilities of being selected for crossover
- genetic operators (i.e., crossover and mutation operators)? their probability of happening?
    - We are using Intermediate arithmetical for our crossover as we are using real values for our chromosomes
    - For mutation, we are adding a small constant value to the real values, and we handle the case of going out of the search space of the problem with some conditions
    - Side note, we could have used Line arithmetical, but Intermediate arithmetical seemed to produce better results probably as it explores a wider space area
    - crossover probability = 0.8, mutation probability = 0.068
- transitioning to the next generation?
    - for the transitioning to the next population we select 1/mating_factor*len(population) pairs of parents that will do crossover
    - If the crossover probability was reached we apply crossover and replace the parents with the new offspring. If the crossover probability is not met we don't do anything we just keep the old parents
    - After selecting the parents and doing crossovers, we go through the whole population and apply mutation to every individual
- Termination strategy?
    - We are using the following formula to determine when to terminate
    - $$\frac{|F_{avg} - F_{best}|}{|F_{avg}|} \leq \epsilon$$
    - Here we simply terminate if the difference between the best fitness and the average fitness converges to a small value (that we define through experimentation). That means it is not worth continuing the genetic algorithm because the children will probably be the same as the parents
    - Also, we stop after 150 iterations if the termination condition is not met

B. Genetic Algorithm Experiments:

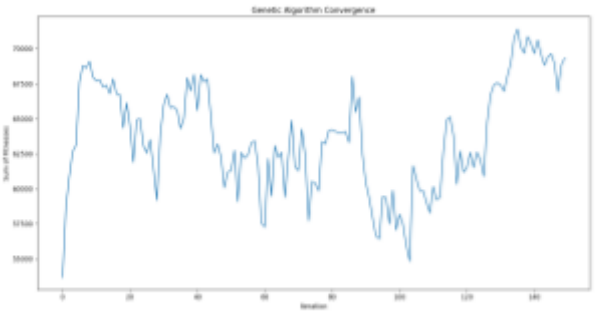
- There were a lot of experiments with the parameters of the genetic algorithm to reach a good setting. In this section instead of going into very technical or mathematical descriptions of how the parameters affect the algorithm as a whole, I will mainly have a quick walkthrough of how the parameters initially started and how by time changed
- Initially, we had the following parameters only for the genetic algorithm
    - incrementing factor = 0.25, mating factor = 2, crossover probability = 0.8
    - current incrementing factor implies we will have a population of (2/0.25)^3 = 512
    - current mating factor implies that ½ * len(population) parent pairs will be selected for mating this means that we will select len(population) parents for mating
    - crossover probability is the chance crossover between parents occurs which is set to 80%
    - Also, at this point, I was using a single-point crossover
- After running the genetic algorithm and observing the overall sum of the fitnesses of the population with time the following were observed
    - The algorithm takes a lot of time to terminate
    - The values of the fitnesses seemed quite random there was no pattern showing that the fitnesses improved over time, and for that reason, I suspected that the crossover technique I used was like a random walk in the search space
- The following actions were taken
    - changed the crossover technique and used Line arithmetical
    - reduced the population by having an incremting factor = 0.4 this implies a population size = 125
- After applying the above changes the fitness value of the population over time increased implying that the genetic algorithm was actually evolving the population and improving it with time the following graph illustrates this behavior. Also, there are 3 main important things to observe from the graph:
    - 1st observation is that the sum of fitnesses increases very quickly in the first 50 iterations
    - 2nd observation is that the fitness of the population stops showing any considerable change at all after iteration 60
    - 3rd observation is that the curve produced is very smooth, implying that going from one population to the other is done in a very systematic way



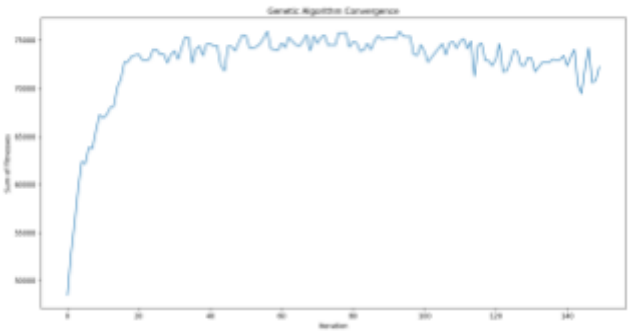
- After having a good crossover technique and good results we needed to apply some mutation to investigate different parts of the search space that our population would not allow us to observe. The mutation technique used is the one discussed in the previous section. Initially, I

set the mutation probability to 50% and had a mutation range of [-0.1, 0.1]. The following graph was produced. There is a single main observation from the graph

- The main observation is that there are very strong fluctuations over the execution of the algorithm, this implies that the sum of fitness for the population with respect to time is not always increasing it keeps going up and down as if we are doing a random search
- For that reason, I kept playing around with the mutation probability until a better graph was produced, and I settled down on a mutation probability of 6.8% and a mutation range = [-0.25, 0.25]

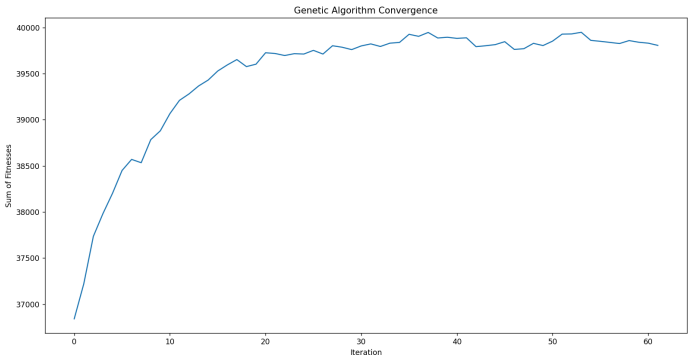


Mutation Probability: 50%



Mutation Probability: 6.8%

- Final parameter adjustments and additional modifications
  - After some experimenting, I decided to go with Intermediate arithmetical as it seemed to produce better classifications. Probably the reason for that is that Intermediate arithmetical explores a wider search space between the 2 parents
  - I have changed the incrementing factor to 0.5 instead of 0.4 which means the population size = 64. This change was for 2 reasons first to have a faster running algorithm, and second, it didn't seem to make much difference between 64 and 125 as population sizes, and probably the reason for that is that the search space for {a0, a1, a2} is quite small
  - Updated the mating factor to 3 instead of 2 which means that at each iteration ⅓*len(population) pairs of parents will be chosen for mating. This change means that the population will evolve at a slower pace and the reason for choosing 3 is that it yields slightly better classification
  - Finally, we have the convergence termination probability as discussed in the previous section we terminate the algorithm if the individuals in the population converge to a specific threshold, the convergence termination probability chosen is 3.1%, and this value was just chosen by trial and error by viewing the graphs produced by the genetic algorithm, the following is an example of execution of the algorithm using all the current parameters we have discussed (the algorithm terminates after around 60 iterations only)



Comparison and Conclusions:

A. Classification Error Comparision:

First, let's start by comparing the classification error for the two approaches. For the K-Nearest Classifier, we will use the average classification error when using k = {7, 19, 31}. For the genetic algorithm, we will use the average classification error for 3 runs of the algorithm

	K-Nearest Classifier	Genetic Algorithm
Average Classification Error (Training)	0.0287	0.1361
Average Classification Error (Validation)	0.05003	0.14259

As we see it seems like the K-Nearest Classifier outperforms the Genetic Algorithm. The following facts hold:

- The K-Nearest Classifier is **4.74** times better at classifying Training Data compared to the Genetic Algorithm
- The K-Nearest Classifier is **2.85** times better at classifying Validation Data compared to the Genetic Algorithm

From the above comparisons, we can conclude that the K-Nearest Classifier outperforms the Genetic Algorithm in our scenario

B. Time Complexity Comparison:

- The Genetic Algorithm roughly has O(nmq) time complexity
- The K-Nearest Classifier roughly has O(m^2) time complexity
    - n: population size
    - m: size of training data
    - q: maximum limit of iterations for the genetic algorithm

- Generally, the K-Nearest Classifier outperforms the Genetic Algorithm in terms of speed for the following reason
    - The Genetic Algorithm has a high constant factor, and by a high constant factor I mean that the Time function of the genetic algorithm might look as follows (dummy example):
        - g(n) = 5*n*m*q + 2*n*q + m where the following terms are dropped from the Big O Notation {5, 2*n*q, m}
    - The time complexity of the K-Nearest Classifier is considered less than the time complexity of the Genetic algorithm if and only if the following condition holds (nq > m)
        - to give some concrete values
        - n = 64
        - m = 360
        - q = 150
        - Genetic Operations = n*m*q = 3,456,000
        - K-Nearest Classifier Operations = m^2 = 129,600

- Given the above concrete values we can say that the K-Nearest Classifier runs roughly 26 times faster than the Genetic Algorithm