Assignment 2  
Anna Bergknut, October

Intro

Genetic algorithms, inspired by natural selection, are invaluable tools for solving complex optimization and machine learning challenges. In this report, we embark on a journey through four tasks.

Task one explores how varying population size and mutation rates impact genetic algorithm performance. This analysis sheds light on their effectiveness in tackling optimization problems.

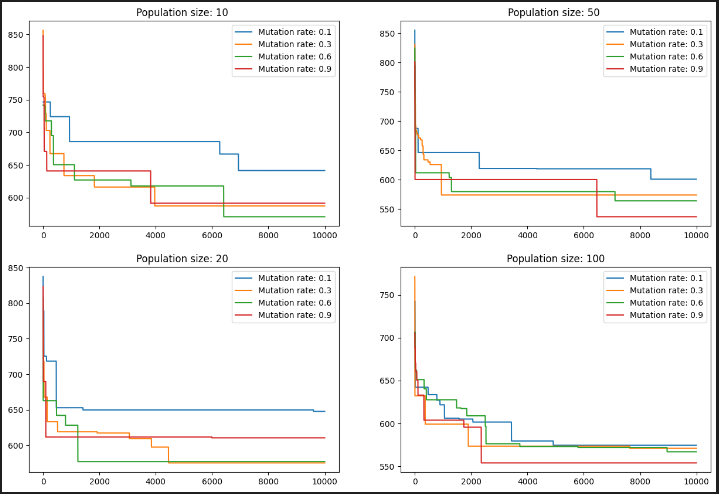
Task two shifts our focus to the application of genetic algorithms in maximizing diverse optimization challenges, revealing their adaptability.

Task three involves developing a custom genetic algorithm for maze navigation, showcasing its ability to find optimal paths and practical utility.

Task four delves into machine learning, comparing linear, polynomial, and neural network models for predictive tasks. The emphasis lies on dataset partitioning for effective model evaluation.

These tasks demonstrate the extensive potential of genetic algorithms in optimization, problem-solving, and machine learning.

Task 1



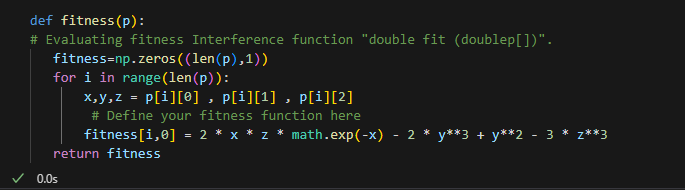
When comparing these results, I can see a clear trend. larger population sizes consistently lead to better average fitness scores. This means that a larger pool of potential solutions improves the algorithm's performance. This is something that is fairly obvious but the graphs clearly indicate it.

Regarding the mutation rate, a noticeable pattern across all graphs is that lower mutation rates resemble slow, incremental steps. While these small steps often yield minor improvements, they contribute to steady progress. In contrast, the population with a higher mutation rate has a more sporadic improvements, but when they occur, they tend to be substantial.

With these observations, I think that a balance is essential. Mutation rates of 0.3 and 0.6 seem to strike this balance well. They consistently demonstrate improvements and maintain reasonable fitness levels.

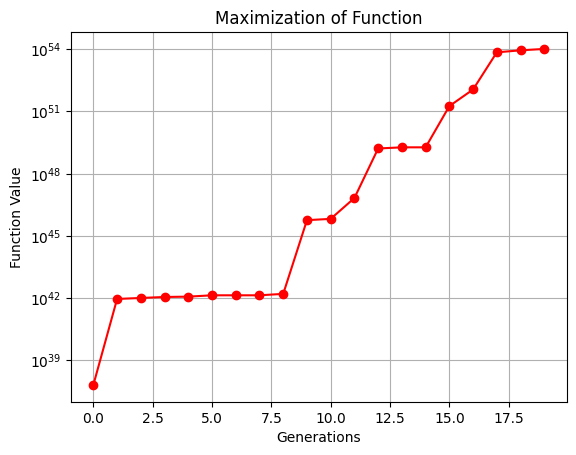
It's important to recognize that genetic algorithms inherently involve an element of randomness. Therefore, the algorithm's performance may variability, but the patterns observed in these results provide valuable insights into its behaviour and potential for optimization.

Task 2



I made changes solely within the fitness function, where I incorporated the mathematical function. Although there might have been room for optimization, my primary focus was on task 3.

The entire code is designed to discover the optimal values of the variables (x, y, z) that maximise a predefined fitness function across multiple generations. The population represents a pool of potential solutions, gradually refining their fitness scores over time. Ultimately, the algorithm tends to yield a reasonably good solution. Unfortunately, this process exhibits considerable variability that i have noticed often when we work with genetic algorithms. The resulting graphs often resemble the graph below.



Task 3

How my code works

I begin by declaring the maze and the variables that I'll need and want easy access to for future changes.

Once the actual code execution starts, I initiate the process by creating the initial population. This is accomplished using a function called create\_individual, which assigns each individual a random path of directions and adds them to a list.

Then, I enter the main loop that will continue until either we discover the endpoint or reach the maximum allowed number of generations.

I promptly evaluate the fitness of each individual. In the funktion evaluate\_fitness , I traverse the paths created by the individuals, and assign a penalty of -0.5 if they collide with walls. My maze structure is built in a way so that each step corresponds to two elements in either the column or row. Therefore, if I check one element in the direction needed and it turns out to be a wall, it signifies that the individual walked into an wall, and I reduce their fitness score. I also deduct -0.2 if an individual backtracks on itself. This is tracked through a list of visited positions during the walk. Because of this list I can also recognize when an individual reaches a new position, where I grant a reward of +0.1. I penalise backtracking more than the reward for discovering a new spot to discourage aimless wandering in the maze.

In fitness evaluation, I also check whether an individual has reached the endpoint. If they have, I add 100 and immediately return the fitness score to halt any further deductions from subsequent steps. Lastly, no matter the result, I return the final fitness score.

Within the genetic\_algorithm function, after assigning fitness scores to all individuals, I identify the best-performing one and check if it has reached the endpoint again. If none have reached the endpoint, I proceed to the selection process for the next generation.

I employ a tournament-style selection approach, where I randomly select 5 individuals and determine which one has the highest fitness. This winner is added to the selected individuals. I repeat this process as many times as needed to fill the population quota for the next generation. Following that, I randomly select parents from the selected population. By combining the two parents' paths, I create a child with a hopefully good path. Then there is a chance to mutate at each step in order to try and find something new and not get stuck in a cycle. This child is added to the new population. I repeat this process until the population size is reached.

This cycle continues until a path is found or until we reach the maximum specified number of generations. Upon discovering the path, the details of the individual that found it are printed, along with a visualisation of the maze that highlights the path.

Improvements

There is some improvements that i would like to implement to make it even better

* Optimising Path Length with fitness

Currently, the algorithm terminates upon finding any path, which may not necessarily be the most efficient one. so if you change it so that it continues. I can modify the fitness function to penalise longer paths, motivating the discovery of shorter, more optimal routes. By incorporating penalties proportional to the number of extra steps taken, you can further encourage the identification of more efficient paths.

* Parallel Processing

The performance is already quite slow. But if I would implement Parallel Processing it would get a lot faster and I could implement heavier bits of code without that big of a concern. Parallelize the fitness evaluation and selection processes. Distributing the workload across multiple processors can significantly speed up the algorithm.

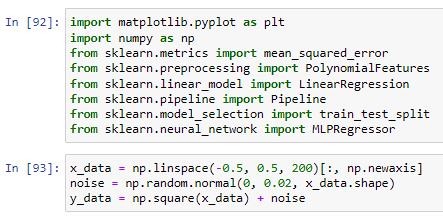
* Advanced Selection Mechanisms

rank-based selection to improve parent selection. Currently it still closes randomly the selected individuals which means that I could pick the 5 worst and the winner gets randomly selected over and over again as parents. So if it was rancid based then we can ensure that it is the top 20% that gets picked. The main problem now is that it would take up a huge amount of time because we are talking about thousands of generations so we would need the Parallel Processing done.

* Dynamic Mutation Rate

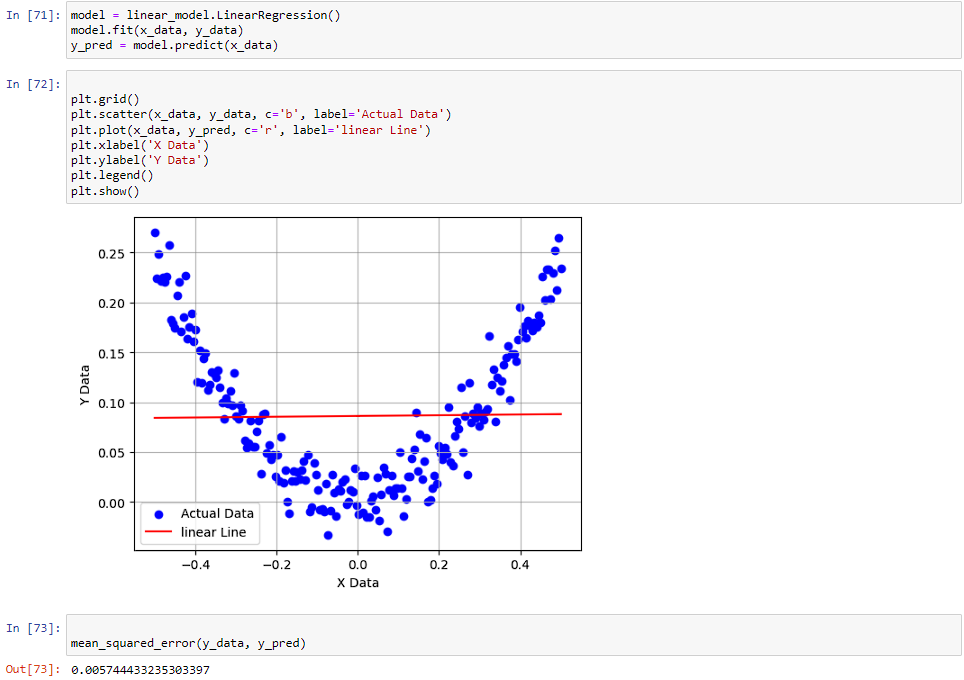
Implement a dynamic mutation rate that increases in the early generations and decreases as the algorithm progresses. This can help strike a balance between exploration and exploitation.

Task 4

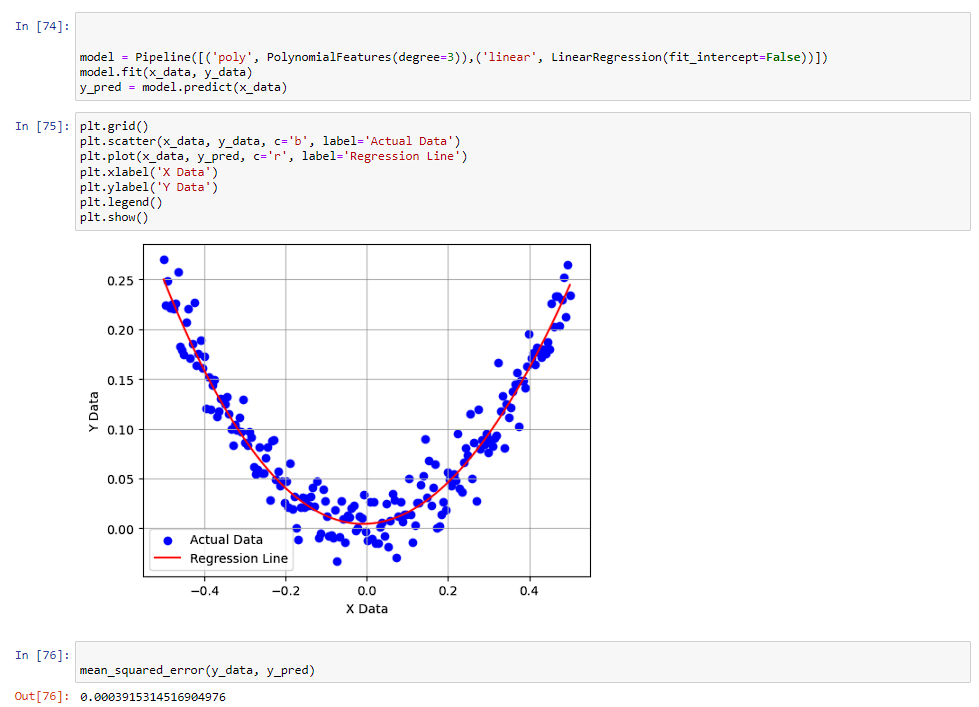


I start by setting up the data and the imports that I will need throughout task 4.

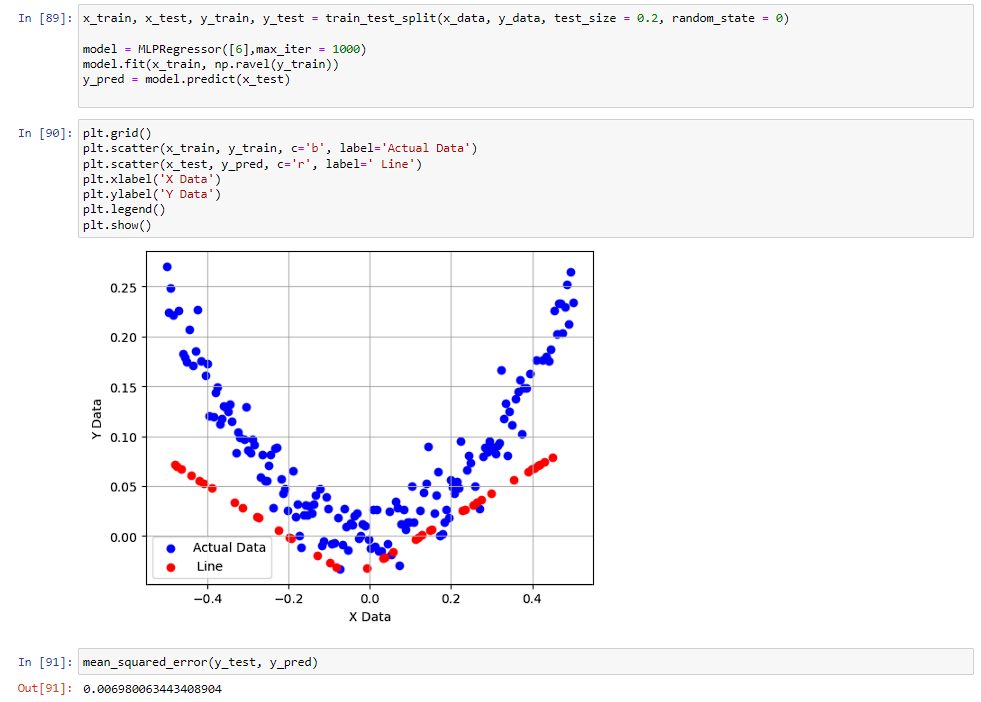
As recommended in the assignment instructions I have been using sklearn library and the website throughout the task.

1. 

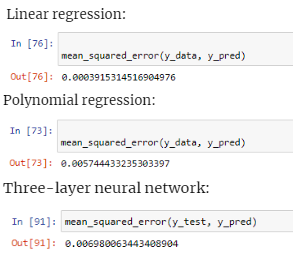
the code establishes a linear regression model, fits it to the given data, and then uses that model to make predictions of 'y' based on new or the same 'x' data. At the end we plot the data together with the actual data to see how it did.

1. 

this code establishes a polynomial regression model with quadratic features to predict 'y' data, which allows for modeling nonlinear relationships between the independent variable ('x\_data') and the dependent variable ('y\_data'). Like last time at the end we plot the data together with the actual data to see how it did.

1. 

The code sets up a three-layer neural network designed for predicting 'y' data. The implementation leverages the MLPRegressor from scikit-learn. The dataset is divided into training and testing subsets, allocating 80% for training and reserving 20% for testing. This separation allows for an effective evaluation of the model's performance. I will say when i ran this function multiple times it widely varied in result. Lastly much like the others we visualise it with a graph.

1. 

After each model I calculate the squared errors interestingly the squared errors are the smallest on the linear model. i don't know why.

1. I had already done that in the start in order to see if the code was working.