

APPLIED AI LAB 2

Morris Simons

November 24, 2023

Abstract

This abstract provides an overview of Assignment 2 in the DV2618/DV2619 course titled "Genetic Algorithm, Data Science and Implementations." The assignment consists of four distinct tasks aimed at applying knowledge of Genetic Algorithms (GAs), Graph Search, Data Science, and implementation skills, preferably using Python. The assignment draws upon concepts covered in course lectures 4, 5, 6, and 7. The deadline for submission is October 9, 2023, at 23:59 pm, and submissions are to be made through the Canvas course webpage with a specific naming convention.

Task 1 focuses on solving the Traveling Salesman Problem (TSP) using a GA. The TSP involves finding the shortest route to visit a set of locations while minimizing travel costs and distance. Students will explore the impact of population size and mutation probability on the GA's performance, with parameter values ranging from 10 to 100 for population size and 0.1 to 0.9 for mutation rate. The results will be analyzed and discussed in the report.

Task 2 involves solving a maximization problem by implementing a fitness function for a given mathematical equation. The GA will optimize the equation, and students must include the fitness function implementation and submit their Python code alongside the report.

Task 3 challenges students to solve a maze problem using a Genetic Algorithm to find the shortest and viable path for a mouse to reach its destination or food within a maze. The maze is represented as a 2D matrix in Python, with possible paths marked as 1 and obstacles as a high value (e.g., 1000). The implementation should be done in Python and submitted as a .py or .ipynb file.

Task 4 requires the use of Python software to perform regression analysis on generated data points. Students will establish linear regression, polynomial regression (power of 2), and a three-layer neural network to predict data points. The dataset is split into training and testing sets, and mean squared errors for the three models are compared. Data points and prediction curves are visualized in the report.

In conclusion, this assignment assesses students' understanding and application of Genetic Algorithms, data science, and implementation skills. The evaluation criteria for Assignment 2 can be found on the Canvas course webpage, and students are encouraged to adhere to academic integrity guidelines.

1 task 1

1.0.1 Collecting data for task 1

first i wanted to automate running all the tests so i did so by adding all the parameter values into a list and then looping them. When done i wanted it to write the text into a txt folder. After the test where done i started doing some data cleaning by converting the data format into json for easier handling.

1.0.2 Viz of data

now we try ordering the data in different ways to maybe find some patterns. By first glance we cannot find any clear patterns that stands out in a obvious way. So we try to visualize the data in another way to understand it at an deeper level.

I made some attempts to visualize the data with minimal success in getting an deeper understanding of the data.

```

population = [10, 20, 50, 100]
mutation_rates = [0.9, 0.6, 0.3, 0.1]
for n_population in population:
    for mutation_rate in mutation_rates:
        n_cities = 20
        #Doing code...
        timer = datetime.now().strftime("%d/%m/%y %H:%M")
        best_solution = [-1,np.inf,np.array([])]
        for i in range(10000):
            if i%50==0: print(i, best_solution[1], fitness_list.mean(), datetime.now().strftime("%d/%m/%y %H:%M"))
            fitness_list = get_all_fitness(mutated_pop,cities_dict)

            #Saving the best solution
            if fitness_list.min() < best_solution[1]:
                best_solution[0] = i
                best_solution[1] = fitness_list.min()
                best_solution[2] = np.array(mutated_pop)[fitness_list.min() == fitness_list]

            progenitor_list = progenitor_selection(population_set,fitness_list)
            new_population_set = mate_population(progenitor_list)

            mutated_pop = mutate_population(new_population_set)
        #stop time stamp
        timer = datetime.now() - datetime.strptime(timer, "%d/%m/%y %H:%M")
        print(f"time to execute: {timer}")
        # %%

        with open(f"data.txt", "a") as file:
            file.write(f"{n_population}: mutationrate: {mutation_rate} Logged {best_solution}: time: {timer}\n")

        print(f"[+] {n_population}: mutationrate: {mutation_rate} Logged {best_solution}: time: {timer}")

```

Figure 1: This is the code i used to collect the data.

popultaion	mutation	gen	Distance
10	0.9	77	606.79
10	0.6	3519	619.52
10	0.3	4744	555.90
10	0.1	5618	655.86
20	0.9	4949	498.61
20	0.6	9956	586.38
20	0.3	5160	564.88
20	0.1	2009	564.34
50	0.9	5195	572.86
50	0.6	5505	533.06
50	0.3	8395	593.81
50	0.1	4768	583.95
100	0.9	3569	615.82
100	0.6	2457	448.60
100	0.3	9737	512.43
100	0.1	8844	538.54

Table 1: This is all the data. in collected order

Table 2: Global caption

Table 3:				Table 4:			
popultaion	mutation	gen	Distance	popultaion	mutation	gen	Distance
100	0.6	2457	448.60	10	0.9	77	606.79
20	0.9	4949	498.61	20	0.1	2009	564.34
100	0.3	9737	512.43	100	0.6	2457	448.60
50	0.6	5505	533.06	10	0.6	3519	619.52
100	0.1	8844	538.54	100	0.9	3569	615.82
10	0.3	4744	555.90	10	0.3	4744	555.90
20	0.1	2009	564.34	50	0.1	4768	583.95
20	0.3	5160	564.88	20	0.9	4949	498.61
50	0.9	5195	572.86	20	0.3	5160	564.88
50	0.1	4768	583.95	50	0.9	5195	572.86
20	0.6	9956	586.38	50	0.6	5505	533.06
50	0.3	8395	593.81	10	0.1	5618	655.86
10	0.9	77	606.79	50	0.3	8395	593.81
100	0.9	3569	615.82	100	0.1	8844	538.54
10	0.6	3519	619.52	100	0.3	9737	512.43
10	0.1	5618	655.86	20	0.6	9956	586.38

2 Task 2

2.0.1 Viz of data in fig 3

In the second task, the focus shifted to the rapid implementation of a mathematical function, as delineated in Figure 4. The results derived from this implementation are presented in Figure 3. It is important to note that the output of the program varies with each run; thus, the representation in Figure 3 is just one instance of the output. A consistent observation across different iterations is the incremental increase in the function’s value with each successive generation.

2.0.2 math to code in fig 4

In Figure 4 we can see the solution that was implemented to reach the results seen in Figure 3.

3 Task 4

3.0.1 The code we used in fig 5

The fourth task entailed a relatively uncomplicated procedure of implementing a data generation function. This was followed by the application of libraries from Scikit-Learn and TensorFlow for conducting linear regression, polynomial regression, and constructing a neural network. A comparative analysis of the results in task 4 (d), as shown in Figure 7, underscores the objective of achieving the lowest possible mean value. The results indicated that both the neural network and polynomial regression models performed comparably, with polynomial regression exhibiting a marginally superior performance.

3.0.2 Viz of data in fig 6

Further, task 4 (e) entailed a comparative evaluation of linear and polynomial regression models through graphical representation. As depicted in Figure 6, this comparison elucidates the enhanced capability of polynomial regression in predicting outcomes more effectively than the linear regression model.

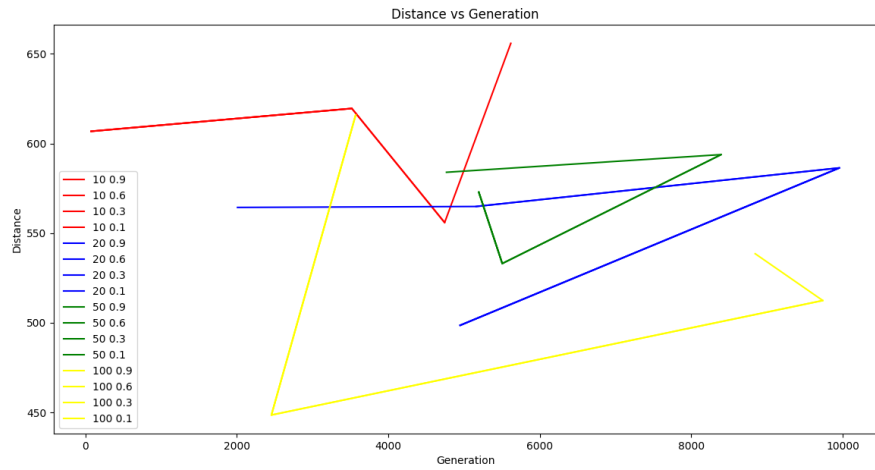


Figure 2: I tried to visualize the data in a more clear way but that did not help

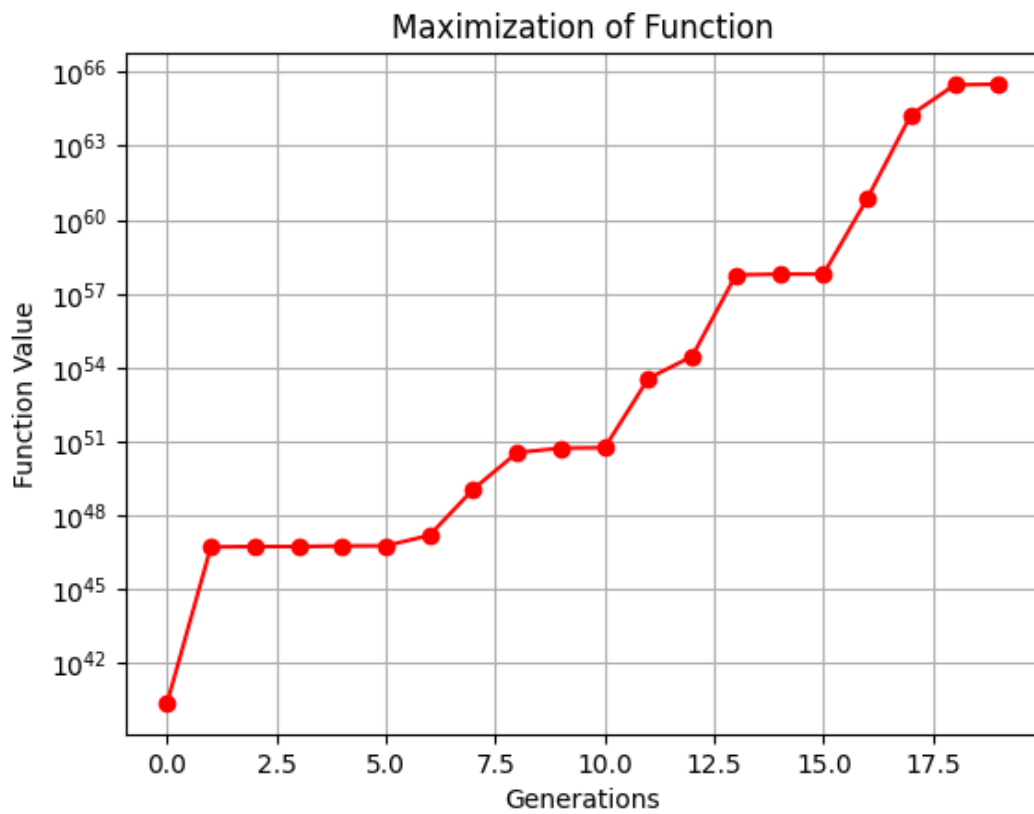
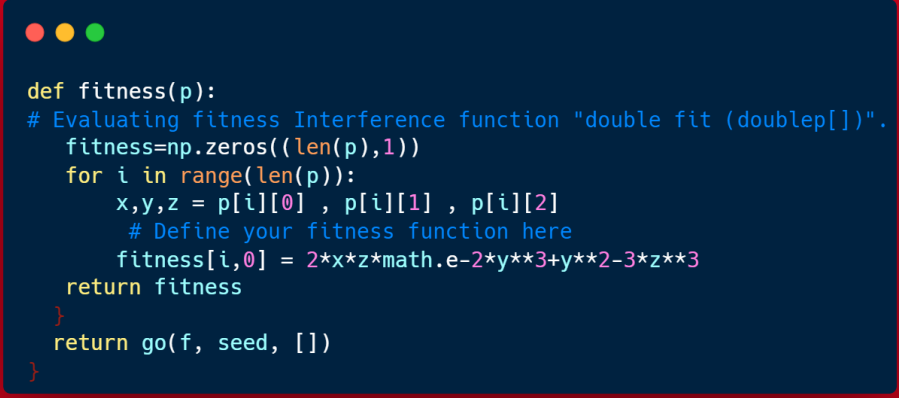


Figure 3: The math function $2 \cdot x \cdot z \cdot e^{-x} - 2 \cdot y^3 + y^2 - 3 \cdot z^3$ in python format



```
def fitness(p):
# Evaluating fitness Interference function "double fit (doublep[])".
fitness=np.zeros((len(p),1))
for i in range(len(p)):
    x,y,z = p[i][0] , p[i][1] , p[i][2]
    # Define your fitness function here
    fitness[i,0] = 2*x*z*math.e-2*y**3+y**2-3*z**3
return fitness
}
return go(f, seed, [])
}
```

Figure 4: this is the math function that we got from the assignment paper

4 Discussion

The initial component of this assignment involved a straightforward task, primarily involving the creation and iteration through a series of lists containing selected values for testing fig 1 to see the code implementation. This process was efficiently executed, with each iteration's outcome being systematically recorded in a tabular format in table 1 and 2. Additionally, there was an effort to visualize the data utilizing the Matplotlib library. However, an analysis of the results did not reveal any discernible patterns between the variables of generations and distance. As seen in fig 2

In the second task, the focus shifted to the rapid implementation of a mathematical function, as delineated in Figure 4. The results derived from this implementation are presented in Figure 3. It is important to note that the output of the program varies with each run; thus, the representation in Figure 3 is just one instance of the output. A consistent observation across different iterations is the incremental increase in the function's value with each successive generation.

The third task of this project did not require any formal reporting, but rather concluded with a final discussion. This task proved to be the most time-intensive, as it involved exploring a variety of options and employing diverse techniques to enhance the model and evaluate the outcomes. At the core of the code lies a crossover function. Initially, I utilized a midpoint crossover approach, which was subsequently replaced with a two-point crossover method. This change also included randomizing the sections where the crossover occurred, thus varying the length and positions of the crossover points each time. The use of the midpoint crossover inherently modified the lower part of the gene sequence.

A notable issue with the midpoint crossover was the increasing homogeneity of genes in later generations. To address this, I developed a new feature termed 'random mutation,' which introduced new gene combinations. This feature, involving a gene that randomly alters some values in the genes, can be deactivated if desired. As the program progresses through each epoch, it was essential to gradually reduce the mutation rate to preserve superior genes. This approach facilitated greater initial experimentation, followed by a more refined mode of operation with a reduced mutation rate in later stages. This was achieved by implementing a learning rate that progressively decreased the mutation rate. Additionally, I set a lower threshold for the mutation rate to ensure it does not diminish to zero, thus sustaining the evolutionary process.

Moreover, I implemented two more techniques to enhance the model. The first involves eliminating a small percentage of the least effective genes, followed by replicating some of the best-performing genes. This approach emerged as one of the most successful in evolving the gene pool.

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def generate_data():
    """Generates data points for the function  $y = x^2 + \text{noise}$ 
    returns x_data(200,1) and y_data(200,1)"""
    x_data = np.linspace(-0.5, 0.5, 200)[:, np.newaxis]
    noise = np.random.normal(0, 0.02, x_data.shape)
    y_data = np.square(x_data) + noise
    return x_data, y_data

def linear_regression(x_train, y_train, x_test, y_test):
    """Linear regression with mean squared error
    args(x_train(160,1), y_train(160,1), x_test(40,1), y_test(40,1))
    returns y_pred_lin(40,1), mse_lin(1,1)"""
    lin_reg = LinearRegression()
    lin_reg.fit(x_train, y_train)
    y_pred_lin = lin_reg.predict(x_test)
    mse_lin = mean_squared_error(y_test, y_pred_lin)
    return y_pred_lin, mse_lin

def polynomial_regression(x_train, y_train, x_test, y_test):
    """Polynomial regression with degree 2 and mean squared error
    args(x_train(160,1), y_train(160,1), x_test(40,1), y_test(40,1))
    returns y_pred_poly(40,1), mse_poly(1,1)"""
    poly_features = PolynomialFeatures(degree=2)
    x_poly_train = poly_features.fit_transform(x_train)
    x_poly_test = poly_features.transform(x_test)
    poly_reg = LinearRegression()
    poly_reg.fit(x_poly_train, y_train)
    y_pred_poly = poly_reg.predict(x_poly_test)
    mse_poly = mean_squared_error(y_test, y_pred_poly)
    return y_pred_poly, mse_poly

def neural_network(x_train, y_train, x_test, y_test):
    """Neural network with 1 hidden layer with 6 nodes and mean squared error
    args(x_train(160,1), y_train(160,1), x_test(40,1), y_test(40,1))
    returns y_pred_nn(40,1), mse_nn(1,1)"""
    model = Sequential()
    model.add(Dense(6, input_dim=1, activation='relu')) # 1 hidden layer with 6 nodes
    model.add(Dense(1, activation='linear')) # Output layer
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(x_train, y_train, epochs=100, batch_size=10)
    y_pred_nn = model.predict(x_test)
    mse_nn = mean_squared_error(y_test, y_pred_nn)
    return y_pred_nn, mse_nn

def plot_results(x_test, y_test, y_pred_lin, y_pred_poly):
    """Plots data points and curve of predictions for Linear and Polynomial Regression
    args(x_test(40,1), y_test(40,1), y_pred_lin(40,1), y_pred_poly(40,1))"""
    plt.scatter(x_test, y_test, color='blue', label='Actual Data')
    plt.plot(x_test, y_pred_lin, color='red', label='Linear Regression')
    plt.plot(x_test, y_pred_poly, color='green', label='Polynomial Regression')
    plt.title('Linear and Polynomial Regression')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    # Generate data
    x_data, y_data = generate_data()
    # Split the dataset into training (80%) and testing (20%) data
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2, random_state=0)
    # (a) Linear Regression
    y_pred_lin, mse_lin = linear_regression(x_train, y_train, x_test, y_test)
    # (b) Polynomial Regression with degree 2
    y_pred_poly, mse_poly = polynomial_regression(x_train, y_train, x_test, y_test)
    # (c) Neural Network
    y_pred_nn, mse_nn = neural_network(x_train, y_train, x_test, y_test)
    # (d) Calculate and compare mean squared errors
    print("Mean Squared Error (Linear Regression):", mse_lin)
    print("Mean Squared Error (Polynomial Regression):", mse_poly)
    print("Mean Squared Error (Neural Network):", mse_nn)
    # (e) Plot data points and curve of predictions for Linear and Polynomial Regression
    plot_results(x_test, y_test, y_pred_lin, y_pred_poly)

```

Figure 5: This is the code used for task 4

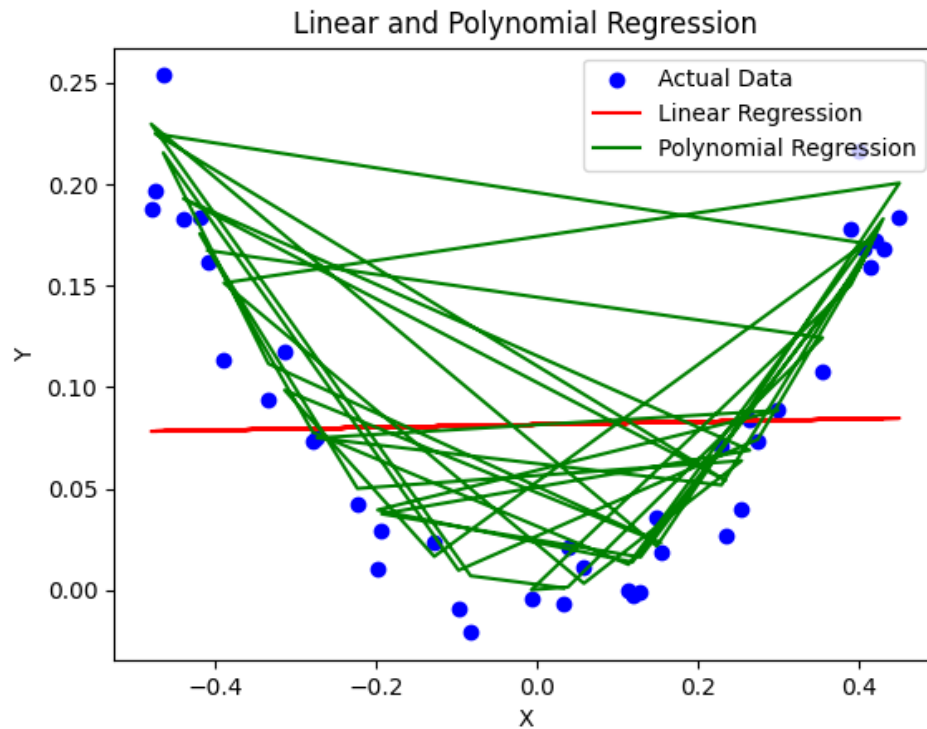


Figure 6: This was the result that we got from running our code in task 4.

```
Epoch 99/100
16/16 [=====] - 0s 867us/step - loss: 5.4070e-04
Epoch 100/100
16/16 [=====] - 0s 933us/step - loss: 5.3661e-04
2/2 [=====] - 0s 1ms/step
Mean Squared Error (Linear Regression): 0.0059784789349812545
Mean Squared Error (Polynomial Regression): 0.00042313387406848254
Mean Squared Error (Neural Network): 0.00047928474346117296
```

Figure 7: And the the printed results.

The fourth task entailed a relatively uncomplicated procedure of implementing a data generation function. This was followed by the application of libraries from Scikit-Learn and TensorFlow for conducting linear regression, polynomial regression, and constructing a neural network. A comparative analysis of the results in task 4 (d), as shown in Figure 7, underscores the objective of achieving the lowest possible mean value. The results indicated that both the neural network and polynomial regression models performed comparably, with polynomial regression exhibiting a marginally superior performance. Further, task 4 (e) entailed a comparative evaluation of linear and polynomial regression models through graphical representation. As depicted in Figure 6, this comparison elucidates the enhanced capability of polynomial regression in predicting outcomes more effectively than the linear regression model.

References