Task1

Report: Traveling Salesman Problem (TSP) Using a Genetic Algorithm (GA)

---

1. Introduction

The Traveling Salesman Problem (TSP) is try to finding the shortest possible route for a salesman to visit a set of cities, visiting each city exactly once and returning to the starting city. The aim is to minimize the total travel distance or cost.here we check the effects of different **population sizes** and **mutation rates** on the performance of the algorithm.

The parameters tested were:

- **Population Sizes**: 10, 20, 50, 100
- **Mutation Rates**: 0.9, 0.6, 0.3, 0.1

---

2. Experimental Setup

The population sizes and mutation rates are the parameter changes others remain same. Each combination of population size and mutation rate was run once, and the results shows following:

- The **mean fitness value** achieved.
- The **iteration or generation number** at which the best solution was found.
- The **path** (sequence of cities) for the best solution.

---

3. Results and Analysis

Mutation Rate	Population Size	Iteration/Generation	Mean Fitness Value	Path (Sequence of Cities)
0.9	100	3697	557.85	['Amsterdam', 'Barcelona', 'Bucharest', ...]
0.6	100	2524	604.54	['Barcelona', 'Berlin', 'Brussels', ...]
0.3	100	7496	482.00	['Sofia', 'Prague', 'Kyiv', 'Warsaw', ...]
0.1	100	5609	550.71	['Barcelona', 'Moscow', 'Vienna', ...]
0.1	50	8387	606.55	['Kyiv', 'Bucharest', 'Milan', ...]
0.3	50	7241	657.41	['Munich', 'Minsk', 'Sofia', 'Kyiv', ...]

Mutation Rate	Population Size	Iteration/Generation	Mean Fitness Value	Path (Sequence of Cities)
0.6	50	8113	573.01	['Barcelona', 'Amsterdam', 'Milan', ...]
0.9	50	394	545.38	['Sofia', 'Munich', 'Birmingham', ...]
0.9	20	6443	566.04	['Munich', 'Brussels', 'Berlin', ...]
0.6	20	7767	570.90	['Paris', 'Vienna', 'Budapest', ...]
0.3	20	3613	536.81	['Vienna', 'Berlin', 'Bucharest', ...]
0.1	20	2277	614.22	['Vienna', 'Bucharest', 'Barcelona', ...]
0.1	10	3780	640.09	['Budapest', 'Vienna', 'Munich', ...]
0.3	10	83	527.56	['Milan', 'London', 'Sofia', ...]
0.6	10	6271	580.12	['Barcelona', 'Prague', 'Amsterdam', ...]
0.9	10	9115	539.50	['Berlin', 'Warsaw', 'Barcelona', ...] link

4. Discussion

4.1 Effect of Mutation Rate

- **Higher mutation rates (0.9, 0.6)** generally result to faster exploration but not necessarily better solutions. For instance, at mutation rate 0.9 and population size 100, the best result was achieved at generation 3697 with a fitness value of **557.85**.
- **Lower mutation rates (0.3, 0.1)** allowed for more controlled exploration, which resulted in the fitness value of **482.00** with a mutation rate of 0.3 and population size of 100,but it happens after many generation (7496).

4.2 Effect of Population Size

Population size also has a effect on the algorithm's performance:

- **Larger populations** (100, 50) provided better results but needs more generations to converge. For example, with population size 100 and mutation rate 0.3, the best result of **482.00** was obtained at generation 7496.
- **Smaller populations** (10, 20) resulted in poorer solutions, but in some cases, the algorithm converged much faster. For instance, with population size 10 and mutation rate 0.3, a solution was found as early as generation 83 with a fitness value of **527.56**.

4.3 Balance Between Population Size and Mutation Rate

The point is that there’s a trade-off between mutation rate and population size. Larger populations with moderate mutation rates (e.g., 0.3, population size 100) produced the best results, presents a balanced approach to searching the solution space.

---

5. Conclusion

The Genetic Algorithm solved the Traveling Salesman Problem across different configurations of population size and mutation rate. The best performance was achieved with a **population size of 100** and a **mutation rate of 0.3**, with a fitness value of **482.00**. This setup found a good middle ground between keeping a variety of genes and finding the best solution. Smaller populations and higher mutation rates showed faster convergence but did not produce as optimal solutions.

Task2

Report: Fitness function implementation for Using a Genetic Algorithm (GA)

---

1. Introduction

The task was to implement a given mathematical function using a Genetic Algorithm. The fitness function includes multiple terms, including exponential and cubic terms, and a cosinus based product. The goal was to find the best values for the variables x, y, and z that maximize the function.

---

2. Fitness Function Implementation

The fitness function takes the variables x, y, and z as inputs and returns the result of the following equation:

Here’s the Python code implementing the function:

Python

```python
def fitness_function(x, y, z):
    x = np.array(x)
    y = np.array(y)
    z = np.array(z)
    term1 = 2 * x * z * np.exp(-x)
    term2 = -2 * y**3
    term3 = y**2
    term4 = -3 * z**3
    dot_product = np.dot(x, z)
    CDP = 1 + np.cos(dot_product)
    expterm = 1 + np.exp(-(np.sum(x + y)))
    exp = CDP / expterm
        result = term1 + term2 + term3 + term4 + exp
    return result
```

This function computes the fitness of individuals based on their values of x, y, and z, I have divided the computation to several term which summed up at last,returning a fitness value that the GA uses to select the best solutions.

---

3. Genetic Algorithm Implementation

The Genetic Algorithm was designed with the following steps:

- **Population Initialization**: Individuals are initialized randomly within the bounds provided for x, y, and z.
- **Selection**: Roulette wheel selection is used to select parents based on their fitness values.
- **Crossover**: One-point crossover is applied between selected parents to generate offspring.
- **Mutation**: Random mutation occurs on one of the genes (x, y, or z) of the offspring with a given mutation probability.

The GA parameters used were:

- Population size: 100
- Number of generations: 50
- Mutation rate: 0.1
- Crossover rate: 0.7

---

4. Results and Analysis

After running the GA, the best solution found was:

- **Best Fitness Value**: 15.5286

- **Values of x**: 3.5981
- **Values of y**: 1.4199
- **Values of z**: -1.8045

---

5. Conclusion

The Genetic Algorithm was effective in optimizing the function by trying to find the best combination of values for x, y, and z.

For Task 4, here is a brief report template summarizing the implementation of regression models and neural networks for data prediction using Python:

Task 4

Report: Regression and Neural Networks for Data Prediction

---

1. Introduction

The task includes generating data points in Python and implementing three models to predict these data points. The models include:

- **Linear Regression**
- **Polynomial Regression (Degree 2)**
- **Three-Layer Neural Network**

Each model was trained and tested using an 80/20 train-test splitting. The models were evaluated based on their mean squared error (MSE), and predictions were presented to compare them to each other.

2. Data Generation

Using NumPy,the x_data and y_data generated based on the following equations:

```
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
```

Here, the x_data consists of 200 points between -0.5 and 0.5. The y_data is generated by squaring x_data and adding a small amount of noise.

3. Model Implementations

3.1 Linear Regression

I implemented linear regression using the LinearRegression class from sklearn.

```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2, random_state=0)
linear_reg = LinearRegression()
linear_reg.fit(x_train, y_train)
linear_pred = linear_reg.predict(x_test)
```

3.2 Polynomial Regression (Degree 2)

For the polynomial regression model, PolynomialFeatures and LinearRegression from sklearn was used.

```
poly_features = PolynomialFeatures(degree=2)
x_poly_train = poly_features.fit_transform(x_train)
poly_model = LinearRegression()
poly_model.fit(x_poly_train, y_train)
y_pred_poly = poly_model.predict(x_poly_test)
```

3.3 Neural Network (Three-Layer Model)

The neural network was implemented using keras with a three-layer structure:

1. Input layer
2. Hidden layer (6 nodes, ReLU activation)
3. Output layer

```
nn_model = Sequential()
nn_model.add(Dense(units=6, activation='relu', input_dim=1))
nn_model.add(Dense(units=1))

nn_model.compile(optimizer='adam', loss='mean_squared_error')
nn_model.fit(x_train, y_train, epochs=300, verbose=0)

y_pred_nn = nn_model.predict(x_test)
```

4. Model Evaluation

The **Mean Squared Error (MSE)** for each model was calculated:

```
mse_linear = mean_squared_error(y_test, linear_pred)
mse_poly = mean_squared_error(y_test_poly, poly_pred)
mse_nn = mean_squared_error(y_test, nn_pred)
```

- **Linear Regression MSE**: mse_linear
- **Polynomial Regression MSE**: mse_poly
- **Neural Network MSE**: mse_nn

Comparing the MSE can show how well each model fits the data.

5. Results

5.1 Linear Regression vs Polynomial Regression (Plot)

the predicted and values for the linear and polynomial regression models:

```
plt.scatter(x_data, y_data, color='blue', label='Data Points', alpha=0.6)  # Plotting original data points
plt.plot(x_plot, y_linear_pred, color='red', label='Linear Regression', linewidth=2)
plt.plot(x_plot, y_poly_pred, color='green', label='Polynomial Regression (degree 2)', linewidth=2)
plt.legend()
plt.show()
```

6. Conclusion

This report demonstrates the implementation of three models for predicting data:

- **Linear Regression** provides a simple but limited fit.
- **Polynomial Regression** captures the non-linear nature of the data better.
- **Neural Network** achieves the most accurate fit, with the smallest MSE, showing its power in modeling non-linear relationships.

The Polynomial Regression model outperforms Linear Regression, while the Neural Network provides the best results for this task.