Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

Link to Github repository: GitHub for Assignment 0

Using a dataset "univariate_profits_and_populations_from_the_cities.csv" and the code sample, the assignment explores how Hyperparameter choices (initial theta values, learning rate (alpha), and number of iterations) would affect the convergence of gradient descent in a linear regression model.

1. Load and Visualize the Data

```
# Load the dataset
data = pd.read_csv('univariate_profits_and_populations_from_the_cities.csv')

# Prepare X_data and y_data
X_data = data['population'].values
y_data = data['profit'].values
m = len(y_data)

# Add a column of ones to X for the bias term θ₀
X = np.c_[np.ones(m), X_data]
y = y_data.reshape((m, 1))
```

This block of code is responsible for loading and preparing the dataset for linear regression. First, the dataset is read from a CSV file using pandas.read_csv() and stored in a DataFrame named data. The dataset contains two key columns: 'population' (the input feature) and 'profit' (the target output). These columns are extracted as NumPy arrays and stored in x_data and y_data, respectively. The variable m is used to store the number of training examples, which is simply the length of the target array y_data.

To prepare the data for matrix operations in linear regression, a column of ones is added to x_data using NumPy's np.c_[] function. This column represents the bias term θ_0, allowing the linear model to include an intercept. The resulting matrix X now has two columns: one for the bias and one for the population values. Lastly, the target array y_data is reshaped into a column vector (y) with shape (m, 1), which ensures compatibility with the linear algebra operations used later in the gradient descent algorithm. This preprocessing step is essential to properly train the linear regression model.

2. I implemented a linear regression model from scratch using NumPy. I defined:
   a. A cost function to evaluate model performance.
   b. A gradient descent algorithm to optimize theta values.

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

```
[30] # Cost function
     def compute_cost(X, y, theta):
         m = len(y)
         predictions = X.dot(theta)
         error = predictions - y
         cost = (1 / (2 * m)) * np.dot(error.T, error)
         return cost.item()

     # Gradient descent
     def gradient_descent(X, y, theta, learning_rate, iterations):
         m = len(y)
         cost_history = []

         for _ in range(iterations):
             prediction = X.dot(theta)
             error = prediction - y
             theta -= (learning_rate / m) * X.T.dot(error)
             cost = compute_cost(X, y, theta)
             cost_history.append(cost)

         return theta, cost_history
```

This part of the code includes two important functions for linear regression: one for calculating the cost and one for performing gradient descent. The first function, compute_cost, measures how well our current model is doing by comparing the predicted values to the actual ones. It does this by multiplying our feature matrix X with the parameter values theta to get predictions, then subtracts the real values y to find the error. The cost is basically the average of the squared errors, which tells us how far off our model is. The smaller the cost, the better the model is performing.

The second function, gradient_descent, helps us find the best values for theta by minimizing the cost function. It does this over a number of iterations. In each step, it calculates predictions, figures out the error, and then updates theta using the gradient descent formula. This formula adjusts the parameters in the direction that reduces the error. After each update, the new cost is calculated and added to a list called cost_history. At the end, the function returns the final theta values and the history of costs so we can see if our model is improving over time.

3. I tested five different configurations, varying:
    c. Initial values of theta
    d. Learning rate
    e. Number of iterations

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

```
[51] # Define parameter combinations to test the impact of Hyperparameters
     combinations = [
         {"theta": np.zeros((2, 1)), "alpha": 0.01, "iters": 1500},
         {"theta": np.zeros((2, 1)), "alpha": 0.001, "iters": 1500},
         {"theta": np.array([[1.0], [1.0]]), "alpha": 0.01, "iters": 500},
         {"theta": np.array([[5.0], [-1.0]]), "alpha": 0.1, "iters": 100},
         {"theta": np.zeros((2, 1)), "alpha": 1.0, "iters": 100},
     ]
```

```
[56] # Run and display results for five different combinations
     for i, combo in enumerate(combinations):
         theta_final, cost_history = gradient_descent(X, y, combo["theta"].copy(), combo["alpha"], combo["iters"])
         print(f"Run {i+1}:")
         print(f"  Initial Theta: {combo['theta'].flatten()}")
         print(f"  Learning Rate: {combo['alpha']}")
         print(f"  Iterations: {combo['iters']}")
         print(f"  Final Theta: {theta_final.flatten()}")
         print(f"  Final Hypothesis: h(x) = {theta_final[0][0]:.4f} + {theta_final[1][0]:.4f}x")
         print(f"  Min Cost: {cost_history[-1]:.4f}")
         print("-" * 50)
```

In this part of the code, I set up five different combinations of parameters to test how they affect the results of gradient descent. Each combination includes an initial guess for the parameter values (theta), a learning rate (alpha), and the number of iterations to run. Some combinations use starting values of zero, while others begin with different numbers to test if the initial guess changes the outcome. The learning rates are also varied from very small (like 0.001) to very large (like 1.0), so I can see how slow or aggressive updates influence convergence. This is important because the assignment requires at least one combination that successfully converges and one that fails, showing the difference between good and bad hyperparameter choices.

In the second block, I loop through each of these combinations and run gradient descent using the specified values. After each run, I print out the key results, like the initial and final theta values, the learning rate used, the number of iterations, the final hypothesis equation, and the minimum cost value. This gives me a clear summary of how each parameter setup performed. For the successful ones, the cost should get smaller and the hypothesis should make sense. For the bad ones, the values might explode or the cost might become infinite, which helps me understand what causes a model to fail. Overall, this section helps demonstrate the impact of tuning parameters in training a linear regression model.

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

I ran five different combinations of hyperparameters to test how they affect the performance of gradient descent in linear regression. In Run 1, I used an initial theta of zeros, a learning rate of 0.01, and 1500 iterations. This setup worked really well, the cost decreased to 4.4834 and the final hypothesis was h(x)=−3.6303+1.1664x, which shows good convergence. Run 2 also converged, but more slowly because the learning rate was smaller (0.001). Even after 1500 iterations, it gave a higher cost of 5.3148 and a less optimal hypothesis.

In Run 3, I tried starting with initial theta values of [1.0, 1.0] and used a learning rate of 0.01 with only 500 iterations. Surprisingly, it still performed well, ending with a cost of 4.8318. This showed that changing the initial theta doesn't hurt much if the learning rate is good and the iterations are enough.

However, Run 4 and Run 5 both failed to converge. In Run 4, I used a higher learning rate of 0.1, and in Run 5, I went even higher to 1.0. Both runs produced huge or infinite values for theta and cost. The final hypothesis equations were unreadable and unrealistic, and the cost in Run 5 was literally inf (infinity). This clearly shows that if the learning rate is too high, the model explodes and becomes useless.

Overall, these tests helped me understand how important it is to choose a proper learning rate. Small changes in hyperparameters can make the difference between a successful model and one that completely breaks.

4. Result Table

Run	Initial Theta	Learning Rate	Iterations	Final Theta	Hypothesis	Final Cost	Converged ?
1	[0.0, 0.0]	0.01	1500	[-3.63, 1.17]	h(x) = -3.63 + 1.17x	4.48	Yes
2	[0.0, 0.0]	0.001	1500	[-0.86,	h(x) =	5.31	Yes

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

				0.89]	-0.8622 + 0.8883x		
3	[1.0, 1.0]	0.01	500	[-1.92, 0.99]	h(x) = -1.9216 + 0.9947x	4.83	Yes
4	[5.0, -1.0]	0.1	100	Huge/diverging values	Diverged	1e173	No
5	[0.0, 0.0]	1.0	100	Huge/diverging values	Diverged	inf	No

- Runs 1–3 successfully minimized the cost.
- Runs 4 and 5 diverged due to overly aggressive learning rates.

```
import pandas as pd

results = []

for i, combo in enumerate(combinations):
    theta_final, cost_history = gradient_descent(X, y, combo["theta"].copy(), combo["alpha"], combo["iters"])
    results.append({
        "Run": i+1,
        "Initial Theta": combo["theta"].flatten().tolist(),
        "Learning Rate": combo["alpha"],
        "Iterations": combo["iters"],
        "Final Theta": theta_final.flatten().tolist(),
        "Hypothesis": f"h(x) = {theta_final[0][0]:.4f} + {theta_final[1][0]:.4f}x",
        "Min Cost": cost_history[-1]
    })

results_df = pd.DataFrame(results)
results_df
```

This block of code is used to collect and summarize the results of running gradient descent with different combinations of hyperparameters. First, the pandas library is imported to enable the use of data structures like DataFrames. An empty list called results is initialized to store the output of each training run.

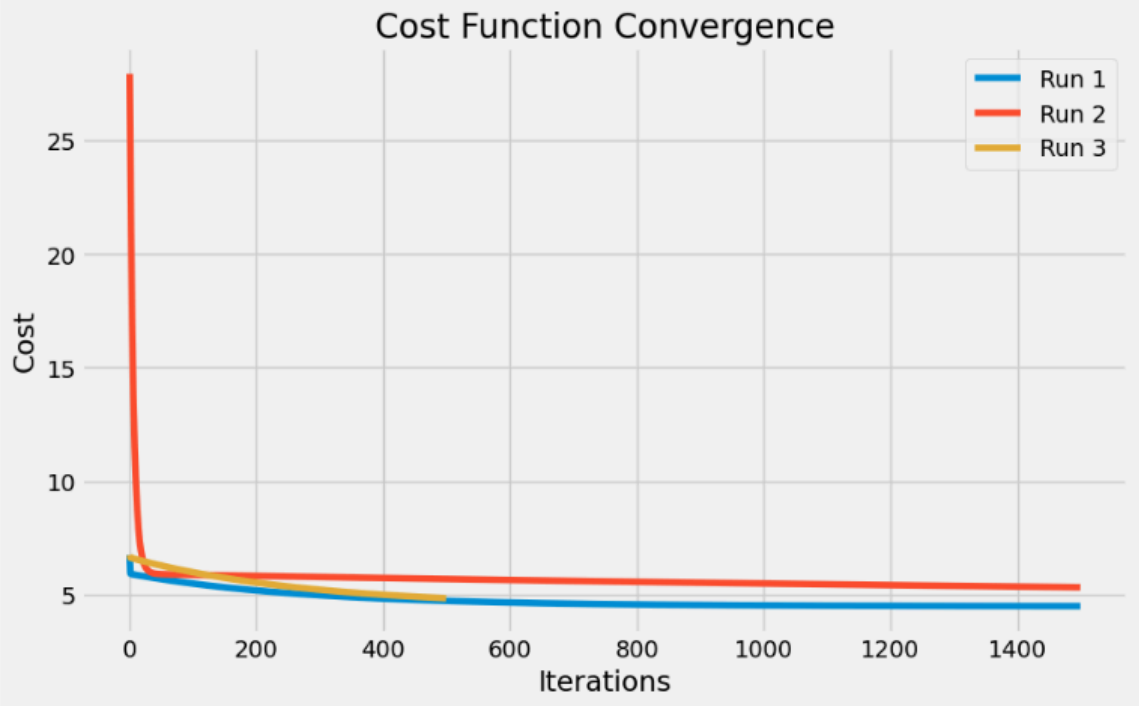
Next, the code enters a loop using enumerate(combinations), where each combo represents a different set of hyperparameters (initial theta, learning rate, and number of iterations). For each combination, the gradient_descent function is executed with the respective parameters. The function returns the final optimized theta values (theta_final) and a list of cost values recorded at each iteration (cost_history).

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

The results of each run are formatted into a dictionary containing key pieces of information: the run number, the initial theta values, the learning rate, the number of iterations, the final theta values, the resulting hypothesis in the form $h(x) = \theta_0 + \theta_1 x$, and the minimum value of the cost function (the last value in cost_history). The theta arrays are flattened and converted to lists for better readability.

Each dictionary is appended to the results list. Once all runs are complete, the list is converted into a Pandas DataFrame named results_df, which displays the summary table. This organized output allows us to easily compare how different settings influenced convergence and model performance.

5. Graph Plotting:




```
[ ]  # Plot Cost Function Convergence
     for i, combo in enumerate(combinations[:3]):
         _, cost_history = gradient_descent(X, y, combo["theta"].copy(), combo["alpha"], combo["iters"])
         plt.plot(range(len(cost_history)), cost_history, label=f"Run {i+1}")

     plt.xlabel('Iterations')
     plt.ylabel('Cost')
     plt.title('Cost Function Convergence')
     plt.legend()
     plt.grid(True)
     plt.show()
```

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 0

This block of code is used to visualize how the cost function decreases over time during gradient descent, helping us evaluate whether or not the model is successfully converging. The loop for i, combo in enumerate(combinations[:3]) iterates over the first three hyperparameter configurations, which are known to converge properly. For each combination, the gradient_descent function is called with the respective theta, learning rate, and iteration count. The cost history returned by the function is a list of cost values calculated at every step of the training process.

Inside the loop, plt.plot() is used to plot the cost values against the number of iterations. Each run is labeled as "Run 1", "Run 2", etc., using label=f"Run {i+1}", which helps distinguish the different lines on the graph. After plotting all three runs, axis labels and a title are added using plt.xlabel(), plt.ylabel(), and plt.title() to clearly define what the graph shows. plt.legend() adds a legend box to identify each curve, and plt.grid(True) makes the graph easier to read. Finally, plt.show() renders the plot.

By limiting the plot to only the first three combinations (which successfully converge), this graph avoids distortion caused by diverging runs (like Run 4 and Run 5, which had extremely large or infinite cost values). This makes it easier to compare how different good parameter settings affect the speed and stability of convergence.

6. Conclusion:

This assignment demonstrated how hyperparameters affect convergence in linear regression. Proper choices of learning rate and iteration count led to smooth cost reduction and stable models. In contrast, high learning rates caused the model to diverge, leading to infinite or nonsensical outputs. I now understand how to debug and tune learning algorithms by monitoring cost behavior and adjusting parameters accordingly.