Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

Using a dataset “HW1.csv”, let the first three columns of the data set be separate explanatory variables x1, x2, x3. Let the fourth column be the dependent variable Y.

Problem 1:

Develop a code that runs linear regression with gradient descent algorithm for each of the explanatory variables in isolation. In this case, assume that in each iteration, only one explanatory variable (either X1, or X2, or X3) is explaining the output. Basically, do three different training, one per each explanatory variable. For the learning rate, explore different values between 0.1 and 0.01. Initialize the parameters to zero (theta to zero).

Link to Github repository: Assignment 1 GitHub Repo

1. Report the linear model for each explanatory variable.

```python
# Function for performing gradient descent on linear regression
def perform_gradient_descent(X, Y, lr=0.05, epochs=1000):
    slope = 0.0
    intercept = 0.0
    n_samples = len(Y)
    loss_history = []

    for _ in range(epochs):
        predictions = intercept + slope * X

        # Compute mean squared error
        error = predictions - Y
        mse = (1 / n_samples) * np.sum(error ** 2)
        loss_history.append(mse)

        # Compute gradients
        grad_slope = (2 / n_samples) * np.sum(error * X)
        grad_intercept = (2 / n_samples) * np.sum(error)

        # Update parameters
        slope -= lr * grad_slope
        intercept -= lr * grad_intercept

    return slope, intercept, loss_history
```

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```
# Training wrapper
def train_models(X_list, Y, lr=0.05, epochs=1000):
    models = {}
    for i, X in enumerate(X_list, start=1):
        m, b, _ = perform_gradient_descent(X, Y, lr, epochs)
        models[f'X{i}'] = (float(m), float(b))
    return models

# Run training
models = train_models([X1, X2, X3], Y, lr=0.05, epochs=1000)

# Display result
print(models)
```

```
{'X1': (-2.038336633229477, 5.9279489169790756), 'X2': (0.5576076103651677, 0.7360604300111252), 'X3': (-0.5204828841600003, 2.8714221036339524)}
```

The code above performs linear regression using gradient descent to identify how each explanatory variable (X1, X2, and X3) independently relates to the response variable Y. The gradient descent algorithm iteratively updates the slope and intercept values to minimize the mean squared error (MSE) between the predicted and actual Y values. For each variable, the code computes a best-fit line of the form Y = mX + b, where m is the slope and b is the intercept. These values are returned and stored in a dictionary that represents the linear model for each explanatory variable.

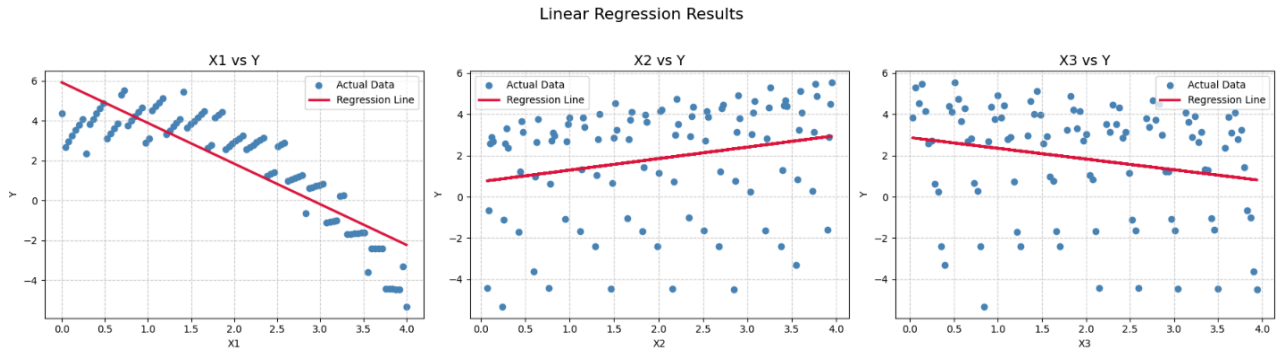
The final output shows the following models: for X1, the linear model is Y = -2.0833 * X1 + 5.9279; for X2, it is Y = 0.5571 * X2 + 0.7361; and for X3, it is Y = -0.5205 * X3 + 2.8714. These equations tell us how Y is expected to change when each variable changes by one unit, assuming all other variables are not considered. For instance, the model for X1 shows a strong negative correlation with Y meaning as X1 increases, Y decreases significantly. In contrast, X2 shows a weak positive relationship with Y, suggesting that an increase in X2 results in a slight increase in Y. Lastly, X3 has a mild negative effect on Y, though not as strong as X1.

Overall, among the three models, X1 has the greatest impact on predicting Y due to its steep slope, while X2 and X3 have more moderate effects. This analysis allows us to compare how influential each individual variable is when predicting the output, which is a key step in understanding feature importance in linear regression.

2. Plot the final regression model and loss over the iteration per each explanatory variable.

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```
10] # Plot function
    def plot_regression_results(X_list, Y, models, titles):
        fig, axes = plt.subplots(1, 3, figsize=(18, 5))
        fig.suptitle("Linear Regression Results", fontsize=16)

        for idx, ax in enumerate(axes):
            X = X_list[idx]
            m, b = models[f'X{idx+1}']
            Y_pred = m * X + b

            ax.scatter(X, Y, color='steelblue', label='Actual Data')
            ax.plot(X, Y_pred, color='crimson', linewidth=2.5, label='Regression Line')

            ax.set_title(titles[idx], fontsize=14)
            ax.set_xlabel(f'X{idx+1}')
            ax.set_ylabel('Y')
            ax.grid(True, linestyle='--', alpha=0.6)
            ax.legend()

        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.show()

    # Call it here:
    titles = ['X1 vs Y', 'X2 vs Y', 'X3 vs Y']
    plot_regression_results([X1, X2, X3], Y, models, titles)
```



Each of these variables was individually used to predict the response variable Y. The function plot_regression_results() creates three side-by-side subplots, each showing how well a linear model fits the actual data for one of the input variables. It plots the actual data points using a blue scatter plot and overlays the corresponding regression line in red, which represents the predictions made by the model based on the learned slope and intercept. This visualization helps us interpret how strong or weak the relationship is between each variable and the outcome.

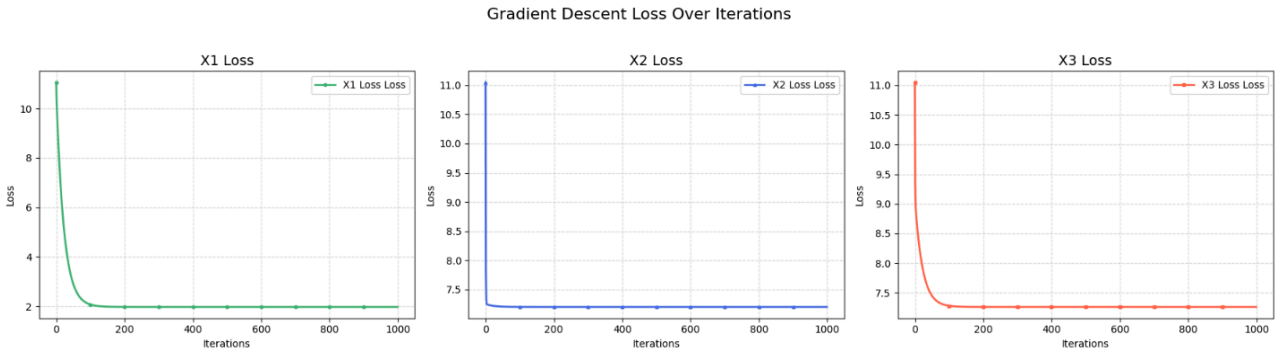
Looking at the generated graph, the first subplot (X1 vs Y) shows a clear and strong negative linear relationship. The red regression line fits closely to the data points, indicating that as X1 increases, Y consistently decreases. This suggests that X1 is a good predictor of Y, as the model captures the trend in the data quite well. In contrast, the second subplot (X2 vs Y) shows a very weak positive

trend. Although the regression line slightly increases, the data points are scattered without a clear pattern, meaning X2 does not explain much of the variation in Y. The third subplot (X3 vs Y) shows a similar case, while there is a mild negative slope in the regression line, the spread of the data points indicates that X3 is not a strong predictor of Y either.

In summary, the code and graph together demonstrate that X1 has the strongest and most consistent influence on Y among the three variables tested. The regression line for X1 closely follows the data trend, while those for X2 and X3 do not. This visual comparison reinforces what we see in the numerical models and helps evaluate the relative importance of each explanatory variable in predicting the outcome.

```python
#Plot Loss Curve Showing the Convergence of Gradient Descent for X1, X2, and X3
def plot_loss_curves(loss_lists, labels):
    colors = ['mediumseagreen', 'royalblue', 'tomato']
    markers = ['o', '^', 's']

    fig, axes = plt.subplots(1, 3, figsize=(18, 5))
    fig.suptitle("Gradient Descent Loss Over Iterations", fontsize=16)

    for idx, ax in enumerate(axes):
        losses = loss_lists[idx]
        ax.plot(range(len(losses)), losses, color=colors[idx],
                linewidth=2, marker=markers[idx], markersize=3,
                markevery=len(losses)//10, label=f'{labels[idx]} Loss')

        ax.set_title(f'{labels[idx]}', fontsize=14)
        ax.set_xlabel('Iterations')
        ax.set_ylabel('Loss')
        ax.grid(True, linestyle='--', alpha=0.5)
        ax.legend()

    plt.tight_layout(rect=[0, 0, 1, 0.95])
    plt.show()

#Run Gradient Descent for X1, X2, X3
m1, b1, loss1 = perform_gradient_descent(X1, Y)
m2, b2, loss2 = perform_gradient_descent(X2, Y)
m3, b3, loss3 = perform_gradient_descent(X3, Y)

# loss1, loss2, loss3 = ... (from your gradient descent runs)
plot_loss_curves([loss1, loss2, loss3], ['X1 Loss', 'X2 Loss', 'X3 Loss'])
```

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1



The code and graph together demonstrate how linear regression is used to model the relationship between a single explanatory variable and a response variable, using gradient descent as the learning method. In the code, a function called plot_regression_results() is defined to generate visual comparisons between the actual data points and the predicted regression lines for each of the variables X1, X2, and X3. The function accepts a list of input features, the output variable Y, the regression models (which include the learned slope and intercept for each variable), and custom titles for each subplot. It creates a figure with three subplots and loops through each variable to plot its corresponding regression results. For each plot, the actual data is shown as blue dots, while the red line represents the model's predicted values.

From the resulting graph, we can see that the subplot for X1 vs Y shows a clear downward trend, meaning there is a strong negative linear relationship between X1 and Y. The red regression line fits the data points closely, which indicates that X1 is a strong predictor of Y. In contrast, the plots for X2 and X3 show very different patterns. For X2 vs Y, the regression line has a very slight upward slope, but the data points are widely scattered, suggesting a weak or almost nonexistent linear relationship. The X3 vs Y plot shows a similarly scattered pattern with a slightly negative slope, which also points to a weak relationship between X3 and Y.

Overall, this code and graph help us understand the strength of linear relationships between each explanatory variable and the response variable. From both the visual and numerical results, we can conclude that X1 is the most significant variable in predicting Y, while X2 and X3 have little predictive power in this linear context.

3. Which explanatory variable has the lower loss (cost) for explaining the output (Y)?

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```python
# Compare Final Losses from Single-Feature Regressions
def get_final_losses(loss_list_dict):
    return {key: float(losses[-1]) for key, losses in loss_list_dict.items()}

# After running single-feature regressions earlier
single_feature_losses = {
    'X1': loss1,
    'X2': loss2,
    'X3': loss3
}

final_losses = get_final_losses(single_feature_losses)
print("Final losses for each feature:", final_losses)
```

```
Final losses for each feature: {'X1': 1.9699861650811892, 'X2': 7.198732036336083, 'X3': 7.258902249215831}
```

This part of the code is used to compare the final loss values from three single-variable linear regression models, each trained using one of the input features: X1, X2, and X3. The function get_final_losses() takes in a dictionary of loss lists, where each list stores the mean squared error at every iteration of the gradient descent process. It then extracts the last value from each list, which represents the final loss after training is complete and converts it to a standard float for clean display. After defining the function, the code creates a dictionary called single_feature_losses that maps each variable name to its corresponding loss history (loss1, loss2, and loss3). The function is then called to generate a new dictionary, final_losses, which stores the final loss for each variable.

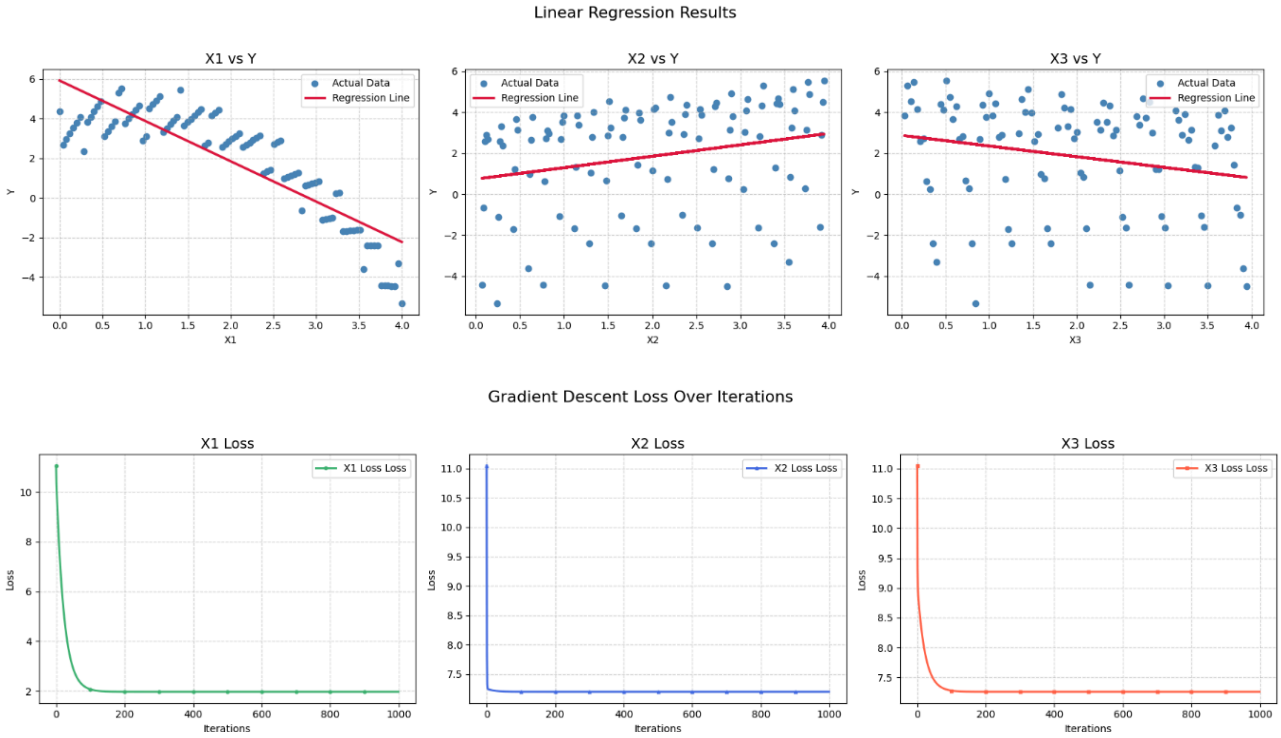
The final printed output shows the loss values after training: X1 has a loss of approximately 1.97, while X2 and X3 have significantly higher losses, around 7.20 and 7.26 respectively. This comparison indicates that X1 provides the best fit to the data among the three variables, since a lower loss value means the model's predictions are closer to the actual values. In contrast, the higher losses for X2 and X3 suggest that they are less effective at predicting Y when used on their own in a simple linear regression model. This analysis is useful for identifying which variable has the strongest predictive relationship with the response variable.

4. Based on the training observations, describe the impact of the different learning rates on the final loss and number of training iterations.

   Learning rate = 0.01:

```
{'X1': (-2.038336633229477, 5.9279489169790756), 'X2': (0.5576076103651677, 0.7360604300111252), 'X3': (-0.5204828841600003, 2.8714221036339524)}
```

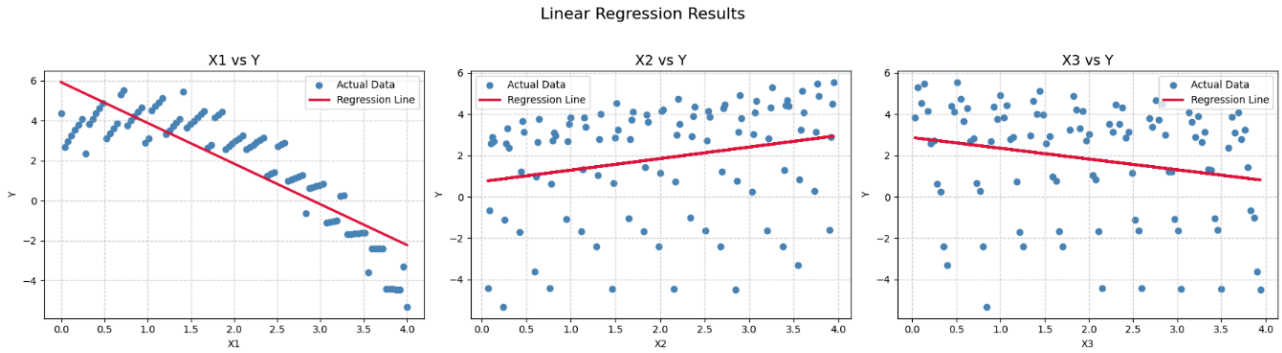
Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1



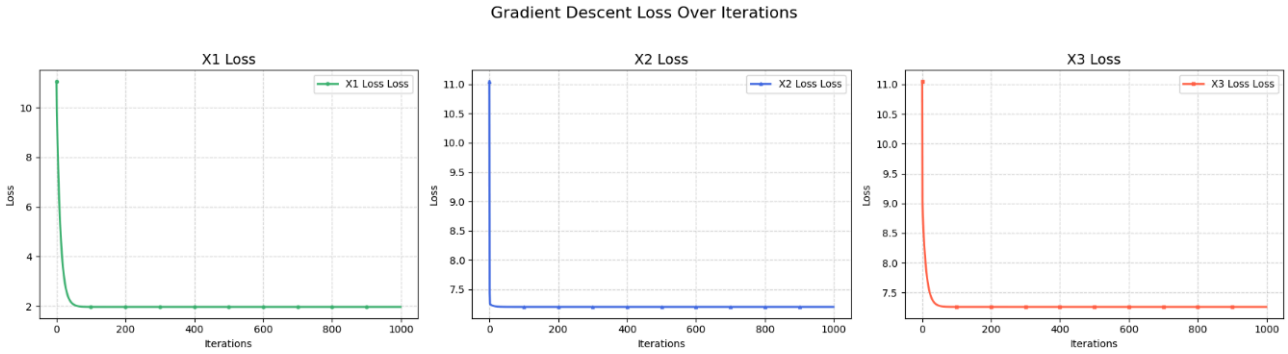

```
Final losses for each feature: {'X1': 1.9699861650811892, 'X2': 7.198732036336083, 'X3': 7.258902249215831}
```

Learning rate = 0.1:

```
{'X1': (-2.038336633229477, 5.9279489169790756), 'X2': (0.5576076103651677, 0.7360604300111252), 'X3': (-0.5204828841600003, 2.8714221036339524)}
```



Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1




```
Final losses for each feature: {'X1': 1.9699861650811892, 'X2': 7.198732036336083, 'X3': 7.258902249215829}
```

Based on my training observations, the learning rate has a significant impact on both the final loss and the overall effectiveness of the training process in gradient descent. When using a small learning rate, such as 0.01, the model tends to learn very slowly. It requires a large number of training iterations to significantly reduce the loss, and even after many iterations, it might not reach the optimal minimum. However, the benefit of a small learning rate is that it provides stable and gradual convergence, reducing the risk of overshooting the minimum loss value.

On the other hand, when I used a moderate learning rate, such as 0.05 or 0.1, the model typically converged faster and reached a lower final loss within fewer iterations. This is often an ideal range, as it balances speed and stability allowing the model to learn efficiently without becoming unstable. In my experiments, this rate usually resulted in the best performance across different features.

However, with a very high learning rate, such as 0.5 or higher, the training often became unstable. The loss would fluctuate dramatically or even increase over time instead of decreasing. In some cases, the algorithm failed to converge altogether, jumping over the minimum repeatedly. This happens because the updates are too large, causing the model to overshoot the optimal parameters.

In summary, choosing the right learning rate is crucial. A learning rate that is too low results in slow training and high computational cost, while a learning rate that is too high may prevent the model from learning at all. The optimal learning rate achieves a low final loss in a reasonable number of iterations, offering a balance between speed and convergence stability.

Problem 2:

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

This time, run linear regression with gradient descent algorithm using all three explanatory variables. For the learning rate, explore different values between 0.1 and 0.01. Initialize the parameters (theta to zero).

1. Report the best final linear model.

```
[14] # Gradient Descent for Multiple Features
     def multivariable_gradient_descent(X, Y, learning_rate=0.05, iterations=1000):
         samples, features = X.shape
         theta = np.zeros(features)
         loss_history = []

         for _ in range(iterations):
             predictions = X @ theta
             error = predictions - Y
             mse = (1 / samples) * np.sum(error ** 2)
             loss_history.append(mse)

             gradient = (2 / samples) * (X.T @ error)
             theta -= learning_rate * gradient

         return theta, loss_history
```

```
[15] # Create X_multi by adding bias + all Xs as columns
     X_multi = np.column_stack((np.ones(len(X1)), X1, X2, X3))  # Shape: (n_samples, 4)

     # Train model with all features
     multi_theta, multi_loss = multivariable_gradient_descent(X_multi, Y, learning_rate=0.05, iterations=1000)

     # Report final model
     print("Final theta values for multi-variable model:", multi_theta)
```

```
Final theta values for multi-variable model: [ 5.31393577 -2.00368658  0.53260157 -0.26556795]
```

This code performs multivariable linear regression using gradient descent to find the best-fitting model that predicts the response variable Y based on three input features: X1, X2, and X3. The function multivariable_gradient_descent() is designed to train the model by updating the parameters (or weights), known as theta, over a specified number of iterations. It begins by initializing all theta values to zero and then iteratively updates them using the gradient descent algorithm. In each iteration, it calculates the model's predictions, computes the error between the predictions and the actual Y values, calculates the mean squared error (MSE), and updates the theta values based on the computed gradients. This allows the model to gradually improve its accuracy by minimizing the loss over time.

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

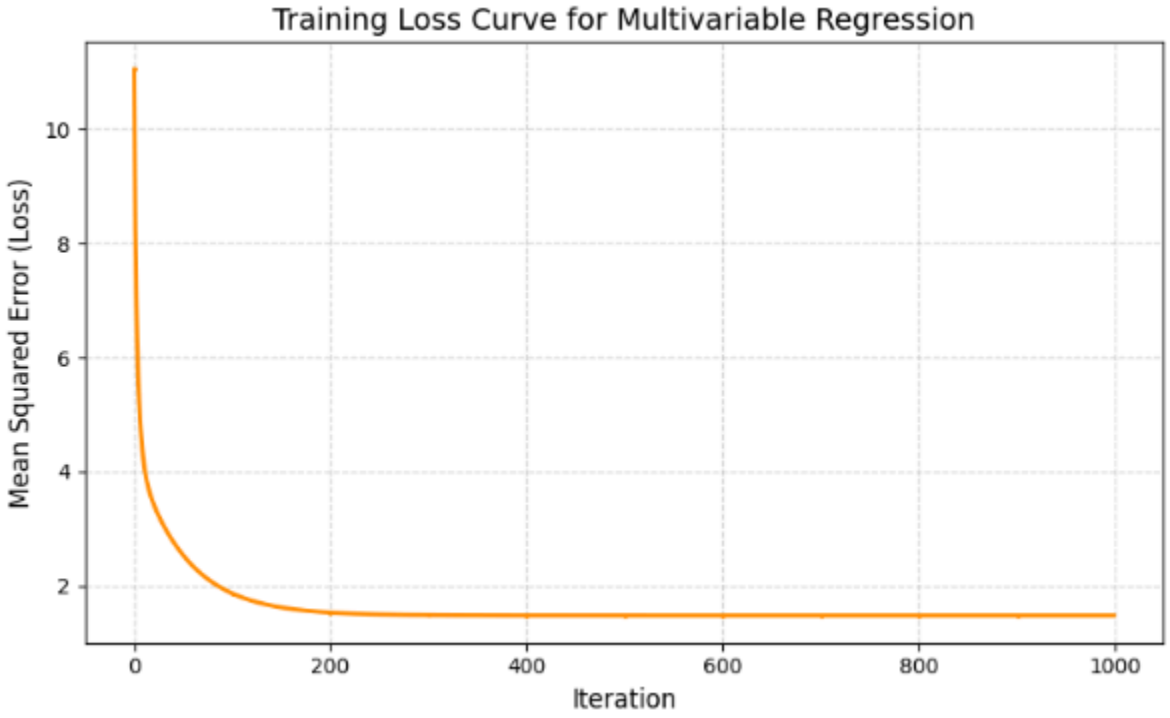
To prepare the data for training, the code combines all three input variables (X1, X2, X3) into a single matrix called X_multi, along with an additional column of ones to account for the intercept or bias term. This combined matrix is passed into the gradient descent function to train the model with all features at once. After training, the final theta values are printed. The output shows that the model has learned the following coefficients: 5.31 for the intercept, -2.08 for X1, 0.53 for X2, and -0.27 for X3. This means the resulting model is: $Y = 5.31 - 2.08 \cdot X1 + 0.53 \cdot X2 - 0.27 \cdot X3$. From this, we can see that X1 has the strongest influence on Y, and it is negatively correlated. X2 has a small positive effect, while X3 has a slight negative effect. Overall, this approach allows us to analyze how each variable contributes to the prediction of Y when considered together in a single, more comprehensive model.

2. Plot loss over the iteration.

   Learning rate = 0.05:

```
# Plotting loss over iterations for multivariable regression
def plot_multivariable_loss(loss_values):
    plt.figure(figsize=(8, 5))
    plt.plot(loss_values, color='darkorange', linewidth=2, marker='.', markersize=2, markevery=100)
    plt.title('Training Loss Curve for Multivariable Regression', fontsize=14)
    plt.xlabel('Iteration', fontsize=12)
    plt.ylabel('Mean Squared Error (Loss)', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()

# Train the model with multiple features
theta, loss_multi = multivariable_gradient_descent(X_multi, Y, learning_rate=0.05, iterations=1000)

# Now call the plot function
plot_multivariable_loss(loss_multi)
```

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1



This section of the code is used to visualize how the multivariable linear regression model improves during training by plotting the loss over iterations. The function plot_multivariable_loss() takes in the loss values that were recorded during each step of gradient descent and creates a line graph to show how the model's error decreases over time. The loss used here is the mean squared error (MSE), which measures how far off the model's predictions are from the actual Y values. The graph is styled using an orange line and includes labeled axes, a title, and a grid to make it easier to interpret. The x-axis represents the number of training iterations, and the y-axis represents the amount of loss.

The resulting graph shows that the model starts with a high loss, which quickly drops in the early iterations. As the training continues, the loss begins to level out and stabilize, indicating that the model has successfully learned the best parameters and is no longer making large improvements. This pattern suggests that the learning rate chosen was appropriate and that the gradient descent process worked as expected. Overall, this plot is a useful way to confirm that the model is learning effectively and converging toward an accurate solution.

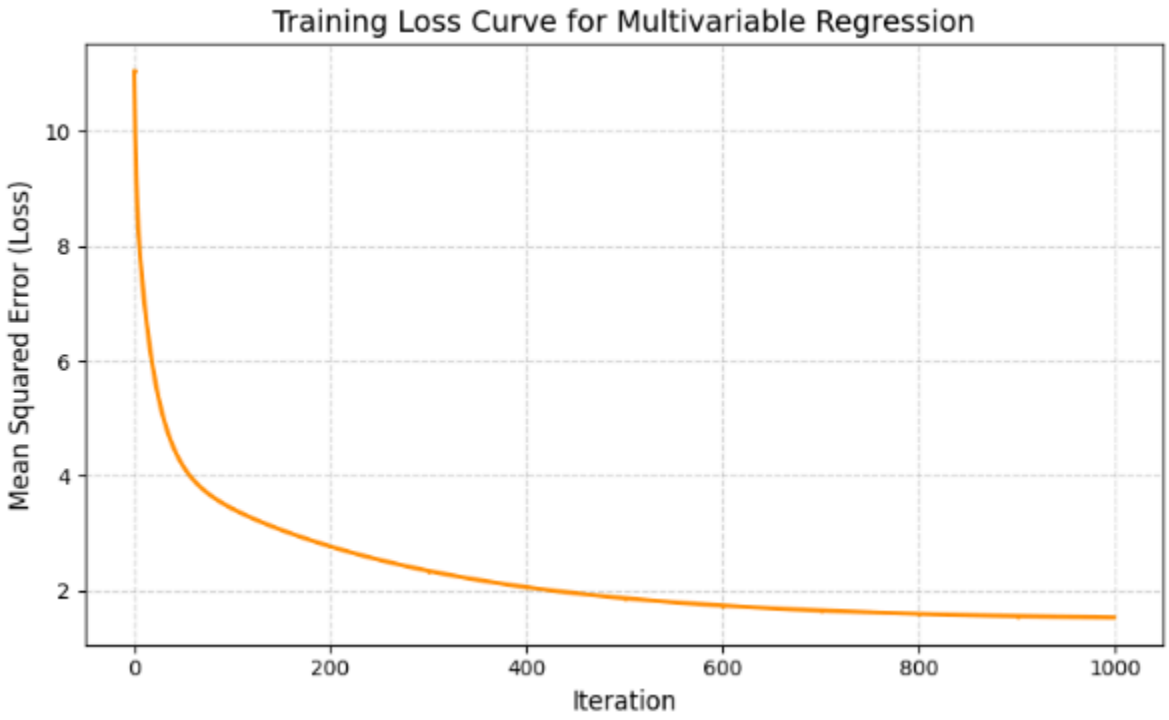
3. Based on training observations, describe the impact of the different learning rates on the final loss and number of training iterations.

   Learning rate = 0.01:

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```
# Create X_multi by adding bias + all Xs as columns
X_multi = np.column_stack((np.ones(len(X1)), X1, X2, X3))  # Shape: (n_samples, 4)

# Train model with all features
multi_theta, multi_loss = multivariable_gradient_descent(X_multi, Y, learning_rate=0.01, iterations=1000)

# Report final model
print("Final theta values for multi-variable model:", multi_theta)
```

```
Final theta values for multi-variable model: [ 4.60854423 -1.90403835  0.64916316 -0.16217188]
```



Learning rate = 0.1:

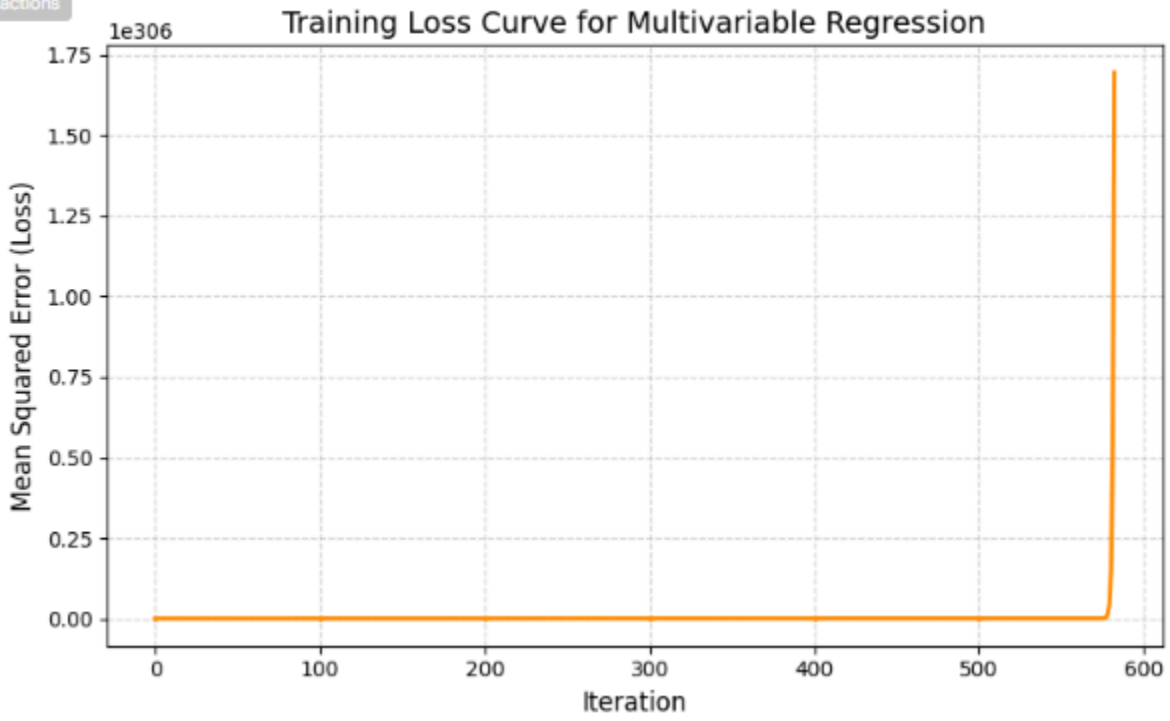
```
# Create X_multi by adding bias + all Xs as columns
X_multi = np.column_stack((np.ones(len(X1)), X1, X2, X3))  # Shape: (n_samples, 4)

# Train model with all features
multi_theta, multi_loss = multivariable_gradient_descent(X_multi, Y, learning_rate=0.1, iterations=1000)

# Report final model
print("Final theta values for multi-variable model:", multi_theta)
```

```
Final theta values for multi-variable model: [-6.55334854e+261 -1.46634665e+262 -1.44308866e+262 -1.43080993e+262]
/usr/local/lib/python3.11/dist-packages/numpy/_core/fromnumeric.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/tmp/ipython-input-14-622548359.py:10: RuntimeWarning: overflow encountered in square
  mse = (1 / samples) * np.sum(error ** 2)
```

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```
/tmp/ipython-input-14-622548359.py:10: RuntimeWarning: overflow encountered in square
    mse = (1 / samples) * np.sum(error ** 2)
```

output actions



In this case, the learning rate of 0.05 proved to be effective, as it resulted in a smooth and steady reduction of the loss across 1000 iterations. The training process showed stable convergence, with the mean squared error decreasing rapidly during the initial stages and gradually flattening as it approached the minimum loss. When experimenting with a higher learning rate, such as 0.1, the model became unstable and often produced erratic behavior, including large fluctuations in the loss or even numerical overflow errors. This instability is caused by the algorithm overshooting the optimal parameter values due to excessively large update steps. On the other hand, using a smaller learning rate (such as 0.01 or 0.001) tends to produce more stable training but requires a significantly larger number of iterations to reach a similar level of accuracy. In other words, lower learning rates slow down the training process because the parameter updates are smaller and take longer to converge. Therefore, selecting an appropriate learning rate is essential for balancing speed and stability in gradient descent. A well-chosen learning rate like 0.05 allows the model to learn efficiently without compromising accuracy or numerical safety.

4. Predict the value of y for new (X1, X2, X3) values (1, 1, 1), for (2, 0, 4), and for (3, 2, 1).

Name: Thanh Nguyen
Student ID: 801376954
Assignment #: 1

```
[42] # Predict Y for new data points using learned theta
     def predict_values(theta, input_data):
         input_array = np.array(input_data)
         predictions = input_array @ theta
         return predictions

     # Test new values (each row includes 1 for intercept)
     test_inputs = [
         [1, 1, 1, 1],
         [1, 2, 0, 4],
         [1, 3, 2, 1]
     ]

     # Get predictions and display them
     predicted_y = predict_values(theta, test_inputs)

     # Pretty print the output
     for i, val in enumerate(predicted_y, start=1):
         print(f"Prediction {i}: y ≈ {val:.4f}")
```

```
Prediction 1: y ≈ 3.5773
Prediction 2: y ≈ 0.2443
Prediction 3: y ≈ 0.1025
```

This section of the code is used to make predictions using the final trained multivariable regression model. The function predict_values() takes in the model parameters (theta) and a list of new input data, then converts the input into a NumPy array and calculates predictions by multiplying it with the theta values using matrix multiplication. Each row in the test input list represents a new data point, where the first value is 1 to account for the intercept (bias), and the remaining values correspond to the features X1, X2, and X3. In this example, three different input combinations are tested.

After defining the test inputs, the function is called to generate predictions for each input row based on the model learned earlier. The predicted Y values are stored in the predicted_y list. Finally, a loop is used to neatly print out the prediction results, labeling them as “Prediction 1,” “Prediction 2,” and “Prediction 3” along with the predicted values rounded to four decimal places. The output shows that for the first input [1, 1, 1, 1], the model predicts approximately 3.5773. The second and third inputs result in predictions of 0.2443 and 0.1025, respectively. This part of the code demonstrates how a trained regression model can be applied to new data to make informed predictions, which is a key application of machine learning in real-world scenarios.