5CS037 - Concepts and Technologies of AI. Linear Models for Regression. Implementation of Linear Regression from Scratch.

Prepared By: Siman Giri {Module Leader - 5CS037}

December 12, 2025

1 Instructions

This worksheet contains programming exercises on building Linear Regression machine learning models based on the material discussed from the slides.This is a graded exercise and submission are mandatory. Please answer the questions below using python in the Jupyter Notebook and follow the guidelines below:

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook.
- Dataset used for this session can be downloaded from shared drive.
- Complete the task only using core python library such as Numpy, pandsa, matplotlib etc. •

Cautions!!! There are total 11 To - Do Exercises Complete all.

Figure 1: Starting with Linear Models.

——————— Linear Models are Everywhere!!! ———————-

2 Understanding Design Matrix for Linear Regression: The

Design Matrix (X) organizes the predictor variables in a dataset into matrix form.

Definition:

A design matrix is a matrix of predictors (independent variables) where:

- Each row corresponds to an observation.
- Each column corresponds to a predictor.

Structure:

The structure of the design matrix X is as follows:

$$X = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{pmatrix}$$

- n: Number of observations (rows).
- p: Number of predictors (columns).

- The first column (all 1s) accounts for the intercept term called bias written as w_0.

3 Building Simple Linear Regression from Scratch.

Simple linear regression models the relationship between one independent variable (x) and one dependent variable (y).

Model:

$$y_i = w_0 + w_1 x_i + \epsilon_i$$

where:

- y_i: Dependent variable for the i-th observation.
- x_i: Independent variable for the i-th observation.
- w_0, w_1: Parameters (intercept and slope).
- ϵ_i: Error term (assumed to have zero mean and constant variance).

Matrix Representation:

For *n* observations, the model can be written as:

$$y = Xw + \epsilon$$

n: y_n □□□ □ 2..

Where: □ □ □1 ϵ . ϵ_n

, ϵ =

□□□ x_1 □ □

Paramete □y_1 □□□ □□□

r 2.. □, X 1 x_2 .. .. . □, w □□□ 1 □□□

y = Estimatio y . = □ϵ □.

1 x_n = w_0 w_1 □

The least squares method minimizes the residual sum of squares (RSS):

$$SSE = \sum_{i=1}^{n} (y_i - (w_0 + w_1 x_i))^2.$$

3.1 Implementation from Scratch Step - by - Step Guide:

3.1.1 Step -1- Data Understanding, Analysis and Preparations:

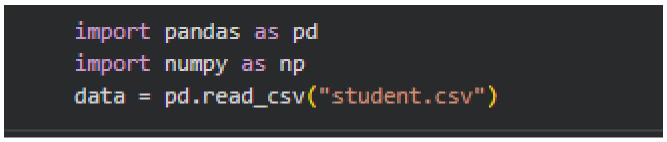
In this step we will read the data, understand the data, perform some basic data cleaning, and store everything in the matrix as shown below.

- Requirements:
  Dataset → student.csv

- Decision Process:
  In this step we will define the objective of the task.

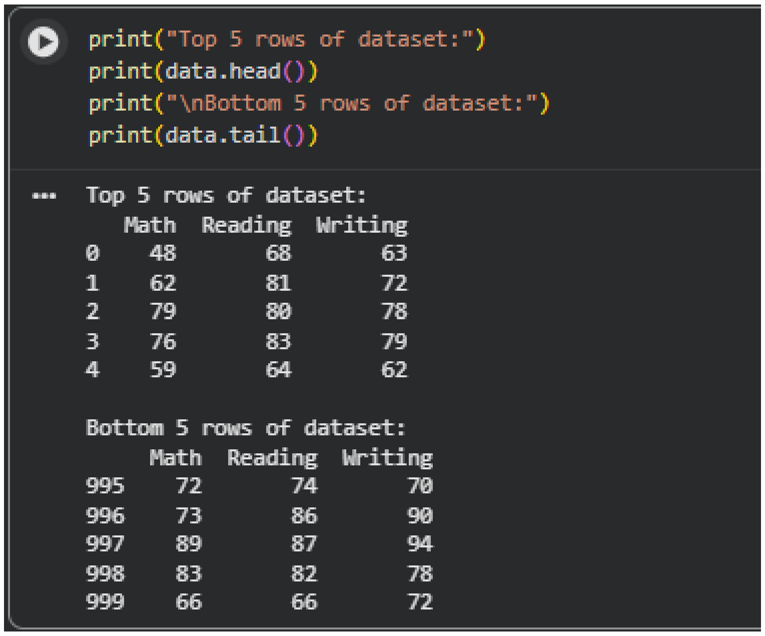
  – Objective of the Task -
  To Predict the marks obtained in writing based on the marks of Math and Reading. • To - Do -

1: 1. Read and Observe the Dataset.



```
import pandas as pd
import numpy as np
data = pd.read_csv("student.csv")
```

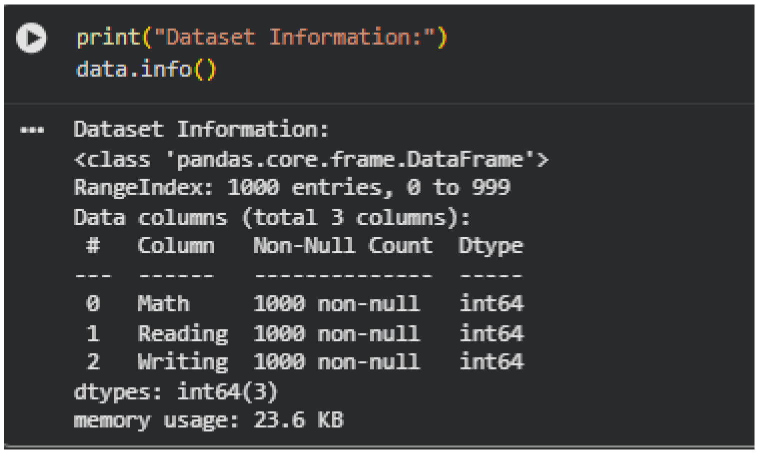
2. Print top(5) and bottom(5) of the dataset {Hint: pd.head and pd.tail}.



```
print("Top 5 rows of dataset:")
print(data.head())
print("\nBottom 5 rows of dataset:")
print(data.tail())
```

```
Top 5 rows of dataset:
   Math  Reading  Writing
0    48       68       63
1    62       81       72
2    79       80       78
3    76       83       79
4    59       64       62

Bottom 5 rows of dataset:
     Math  Reading  Writing
995    72       74       70
996    73       86       90
997    89       87       94
998    83       82       78
999    66       66       72
```

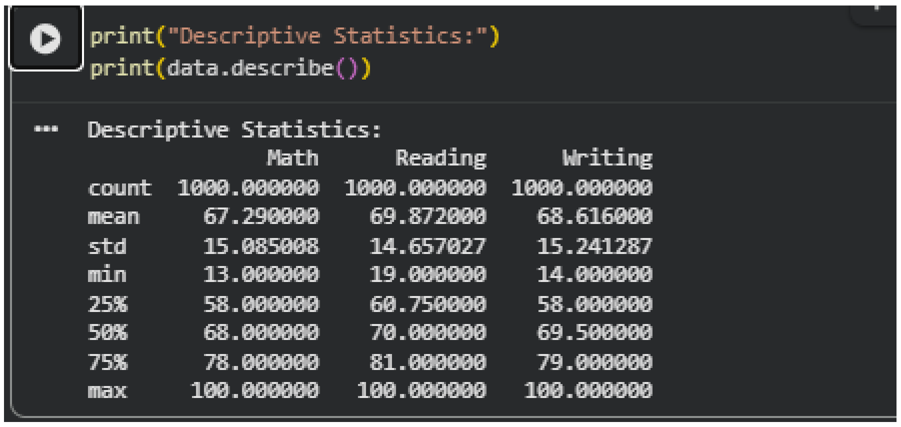
3. Print the Information of Datasets. {Hint: pd.info}.



```
print("Dataset Information:")
data.info()
```

```
Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Math     1000 non-null   int64
 1   Reading  1000 non-null   int64
 2   Writing  1000 non-null   int64
dtypes: int64(3)
memory usage: 23.6 KB
```

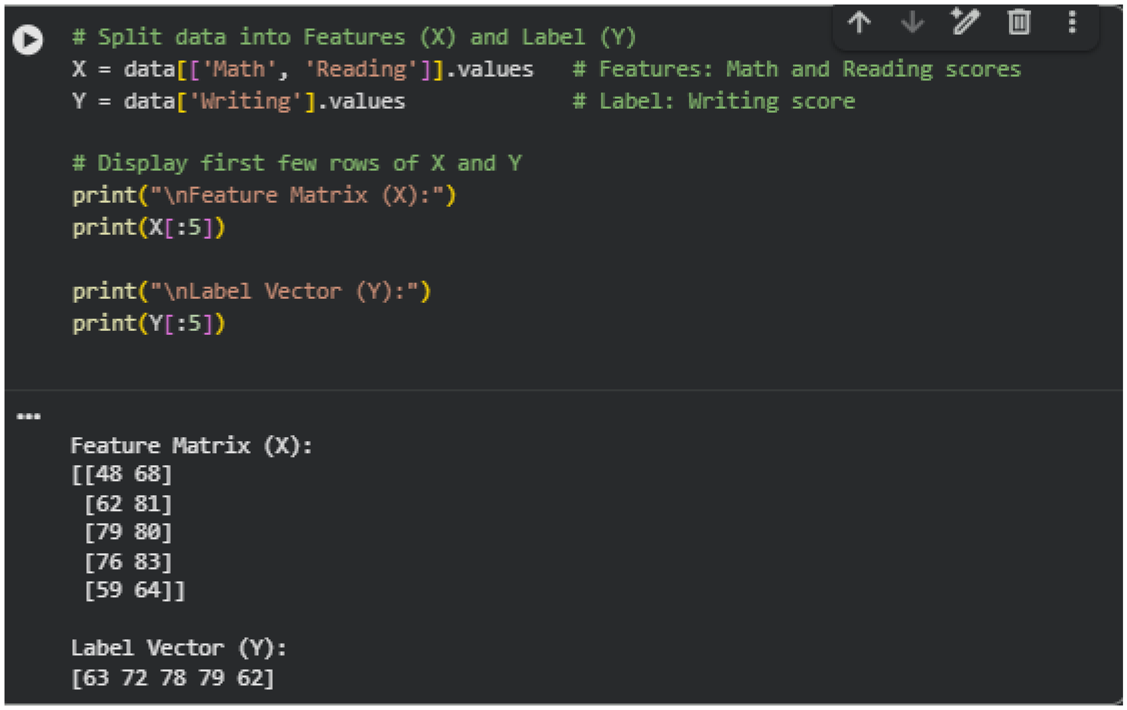
4. Gather the Descriptive info about the Dataset. {Hint: pd.describe}



```
print("Descriptive Statistics:")
print(data.describe())
```

```
Descriptive Statistics:
              Math      Reading      Writing
count  1000.000000  1000.000000  1000.000000
mean     67.290000    69.872000    68.616000
std      15.085008    14.657027    15.241287
min      13.000000    19.000000    14.000000
25%      58.000000    60.750000    58.000000
50%      68.000000    70.000000    69.500000
75%      78.000000    81.000000    79.000000
max     100.000000   100.000000   100.000000
```

5. Split your data into Feature (X) and Label (Y).



```python
# Split data into Features (X) and Label (Y)
X = data[['Math', 'Reading']].values    # Features: Math and Reading scores
Y = data['Writing'].values              # Label: Writing score

# Display first few rows of X and Y
print("\nFeature Matrix (X):")
print(X[:5])

print("\nLabel Vector (Y):")
print(Y[:5])
```

```
Feature Matrix (X):
[[48 68]
 [62 81]
 [79 80]
 [76 83]
 [59 64]]

Label Vector (Y):
[63 72 78 79 62]
```

• To - Do - 2:

1. To make the task easier - let's assume there is no bias or intercept.
2. Create the following matrices:

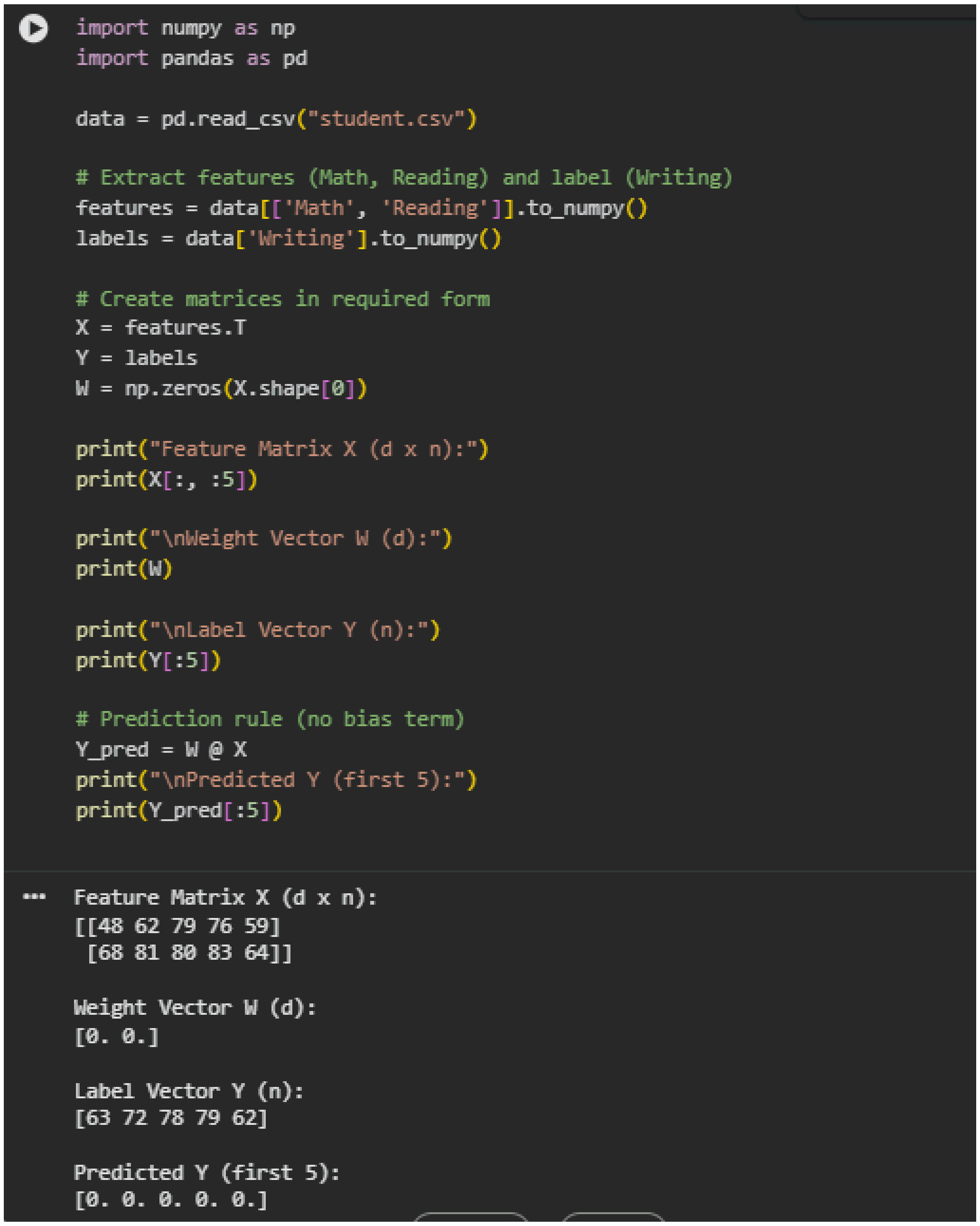
$$Y = W^T X$$

where $W \in \mathbb{R}^d$

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}, \quad X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,n} \end{bmatrix}, \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

where $X \in \mathbb{R}^{d \times n}$, where $Y \in \mathbb{R}^n$

3. Note: The feature matrix described above does not include a column of 1*s*, as it assumes the absence of a bias term in the model.

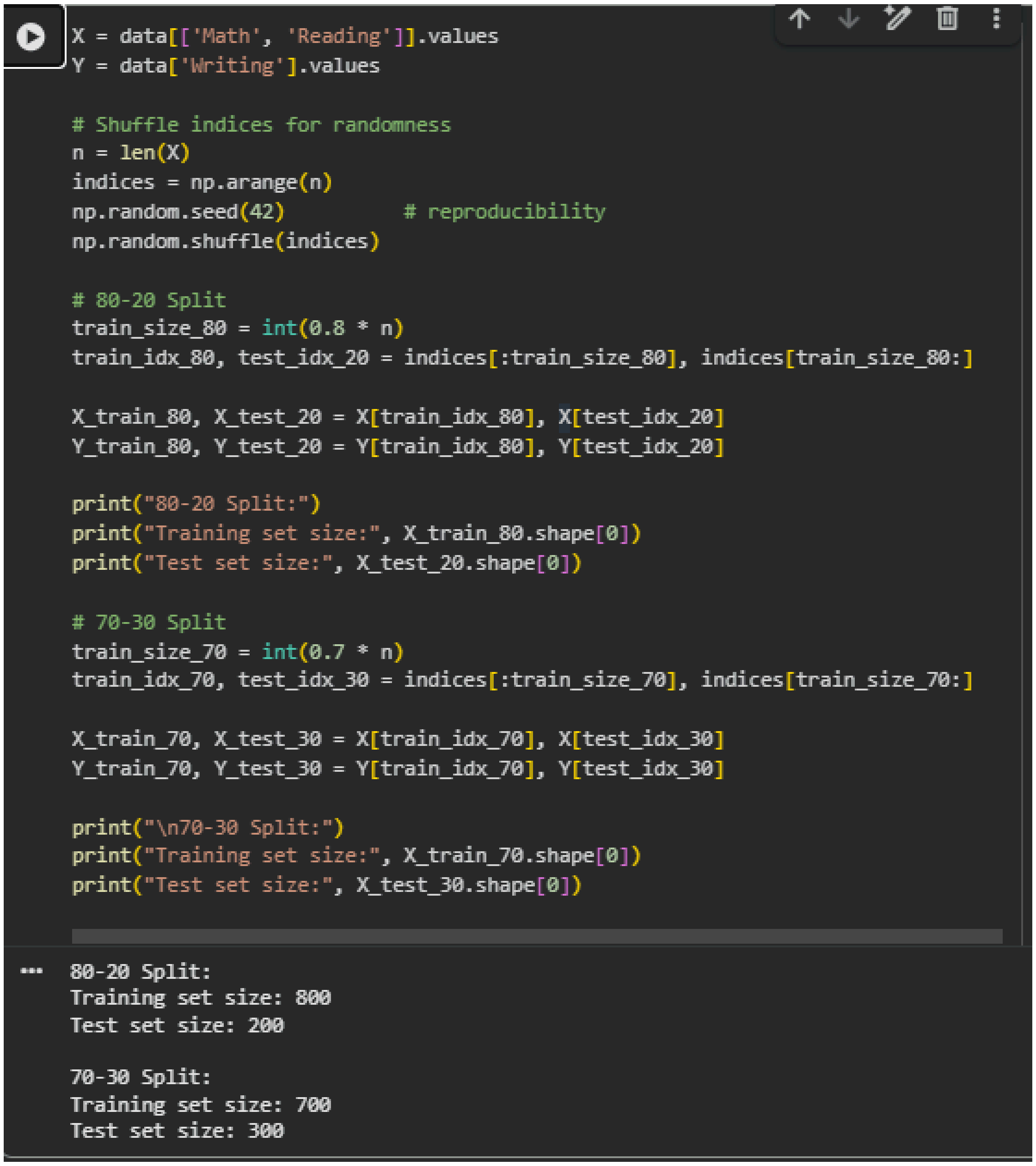


```python
import numpy as np
import pandas as pd

data = pd.read_csv("student.csv")

# Extract features (Math, Reading) and label (Writing)
features = data[['Math', 'Reading']].to_numpy()
labels = data['Writing'].to_numpy()

# Create matrices in required form
X = features.T
Y = labels
W = np.zeros(X.shape[0])

print("Feature Matrix X (d x n):")
print(X[:, :5])

print("\nWeight Vector W (d):")
print(W)

print("\nLabel Vector Y (n):")
print(Y[:5])

# Prediction rule (no bias term)
Y_pred = W @ X
print("\nPredicted Y (first 5):")
print(Y_pred[:5])
```

```
Feature Matrix X (d x n):
[[48 62 79 76 59]
 [68 81 80 83 64]]

Weight Vector W (d):
[0. 0.]

Label Vector Y (n):
[63 72 78 79 62]

Predicted Y (first 5):
[0. 0. 0. 0. 0.]
```

• To - Do - 3:

1. Split the dataset into training and test sets.
2. You can use an 80-20 or 70-30 split, with 80% (or 70%) of the data used for training and the rest for testing.



```
X = data[['Math', 'Reading']].values
Y = data['Writing'].values

# Shuffle indices for randomness
n = len(X)
indices = np.arange(n)
np.random.seed(42)          # reproducibility
np.random.shuffle(indices)

# 80-20 Split
train_size_80 = int(0.8 * n)
train_idx_80, test_idx_20 = indices[:train_size_80], indices[train_size_80:]

X_train_80, X_test_20 = X[train_idx_80], X[test_idx_20]
Y_train_80, Y_test_20 = Y[train_idx_80], Y[test_idx_20]

print("80-20 Split:")
print("Training set size:", X_train_80.shape[0])
print("Test set size:", X_test_20.shape[0])

# 70-30 Split
train_size_70 = int(0.7 * n)
train_idx_70, test_idx_30 = indices[:train_size_70], indices[train_size_70:]

X_train_70, X_test_30 = X[train_idx_70], X[test_idx_30]
Y_train_70, Y_test_30 = Y[train_idx_70], Y[test_idx_30]

print("\n70-30 Split:")
print("Training set size:", X_train_70.shape[0])
print("Test set size:", X_test_30.shape[0])
```

```
80-20 Split:
Training set size: 800
Test set size: 200

70-30 Split:
Training set size: 700
Test set size: 300
```

3.1.2 Step -2- Build a Cost Function:

Cost function is the average of loss function measured across the data point. As the cost function for Regression problem we will be using Mean Square Error which is given by:

$L(w) = \frac{1}{2n} X^n$ $\qquad$ $y_{\text{pred}}(w) = W^T X$

To - Do - 4:

$i=1$

where: $\qquad$ $y_{\text{pred}(i)} - y_i \; 2$

Feel free to build your own code or complete the following code:

Building a Cost Function:

```
#Define the cost function
def cost_function(X, Y, W):
    """ Parameters:
    This function finds the Mean Square Error.
    Input parameters:
      X: Feature Matrix
      Y: Target Matrix
      W: Weight Matrix
    Output Parameters:
      cost: accumulated mean square error.
```

```
    """
    # Your code here:
    return cost
```

Designing a Test Case for Cost Function:

We will first calculate the loss value manually and then verify the output via our code. If the computed value matches, we will proceed further.

Given:

$X = \begin{bmatrix} 1 & 3 \\ 5 & 2 \\ 4 & 6 \end{bmatrix}$, $Y = \begin{bmatrix} 3 \\ 7 \\ 11 \end{bmatrix}$, $W = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

The hypothesis function $h_\theta(X)$ is calculated as: $h_\theta(X) = X \cdot W = \begin{bmatrix} 1 & 3 \\ 5 & 2 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \\ 11 \end{bmatrix}$

Then, the Mean Squared Error (MSE) is calculated as:

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

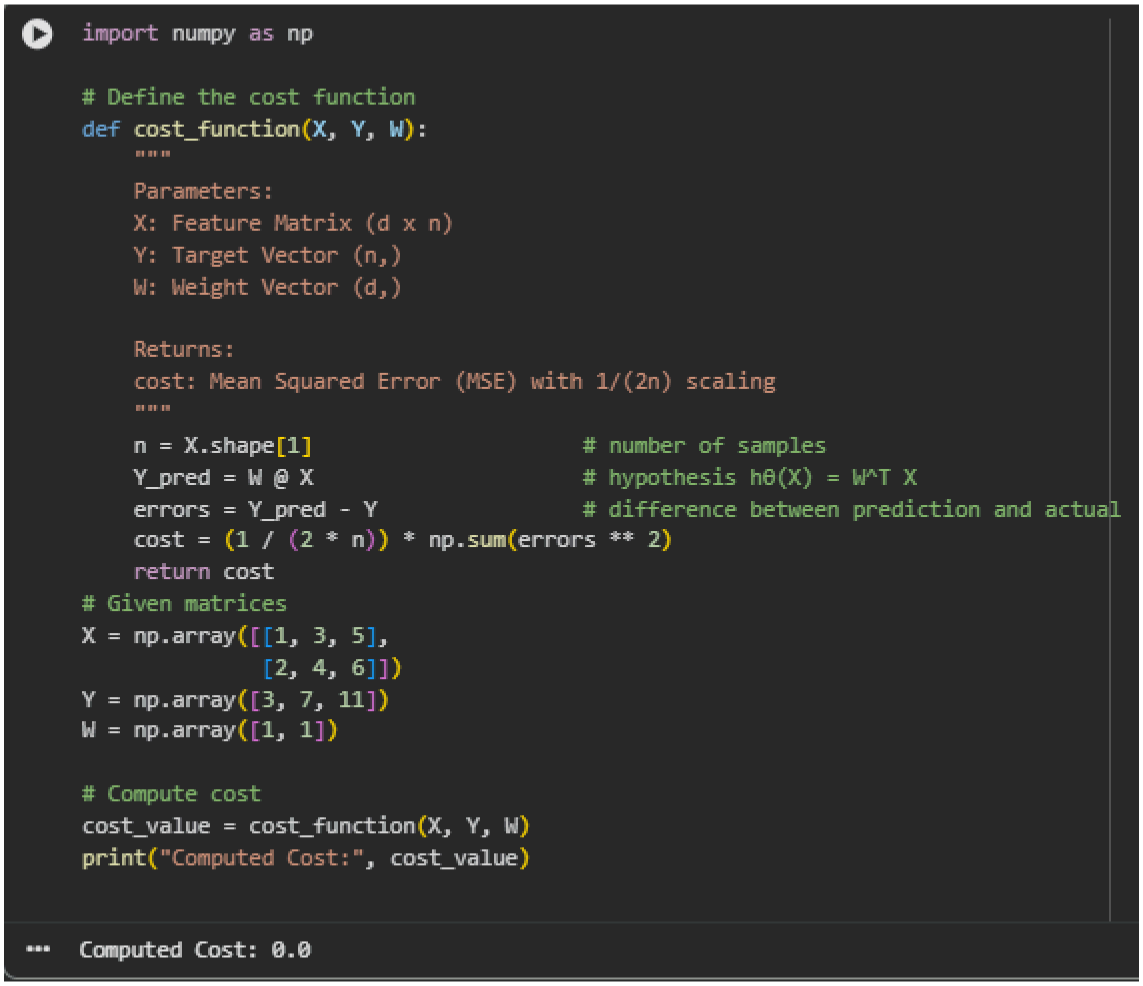
where n is the number of training examples. Substituting the given values:

$$\text{cost} = \frac{1}{6} \left[(3-3)^2 + (7-7)^2 + (11-11)^2 \right]$$

$$\text{cost} = \frac{1}{6} \cdot 0 = 0$$

Thus, for the given test case, the cost function should output:

$$\boxed{0}$$



```python
import numpy as np

# Define the cost function
def cost_function(X, Y, W):
    """
    Parameters:
    X: Feature Matrix (d x n)
    Y: Target Vector (n,)
    W: Weight Vector (d,)

    Returns:
    cost: Mean Squared Error (MSE) with 1/(2n) scaling
    """
    n = X.shape[1]                    # number of samples
    Y_pred = W @ X                    # hypothesis hθ(X) = W^T X
    errors = Y_pred - Y               # difference between prediction and actual
    cost = (1 / (2 * n)) * np.sum(errors ** 2)
    return cost
# Given matrices
X = np.array([[1, 3, 5],
              [2, 4, 6]])
Y = np.array([3, 7, 11])
W = np.array([1, 1])

# Compute cost
cost_value = cost_function(X, Y, W)
print("Computed Cost:", cost_value)
```

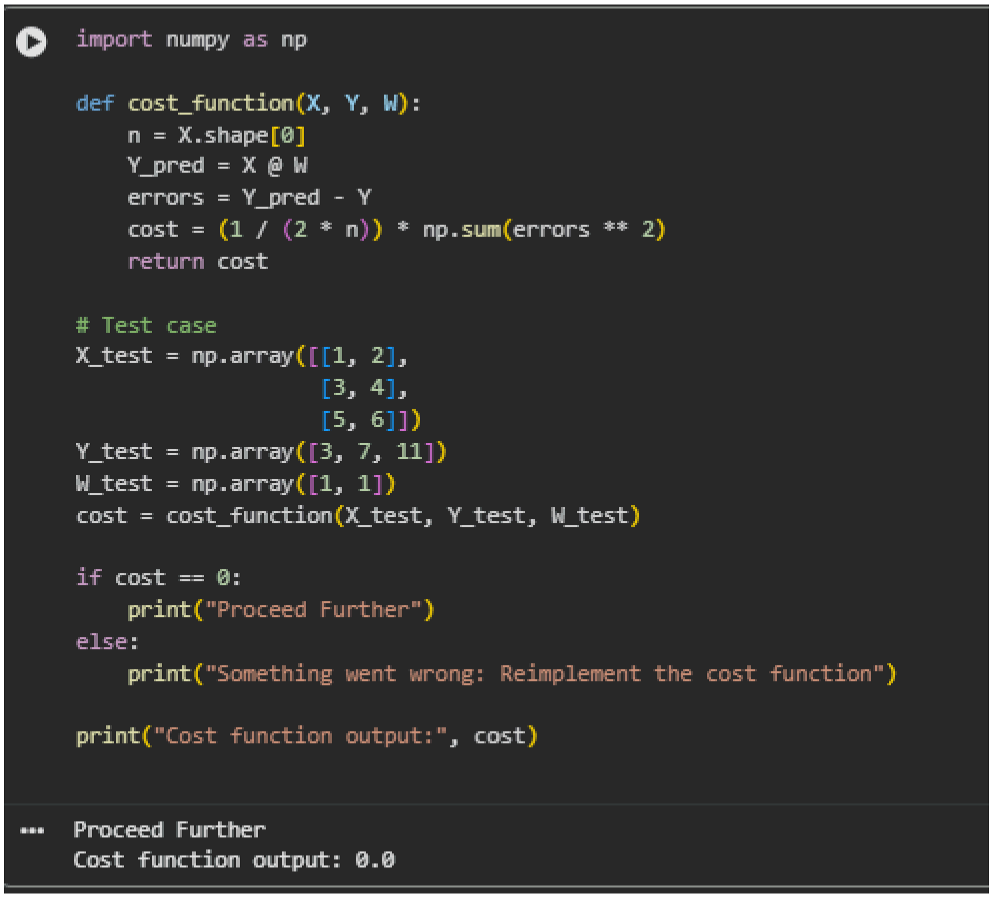
```
Computed Cost: 0.0
```

To - Do - 5:

Make sure your code at To - Do - 4 passed the following test case:

Testing a Cost Function:

```python
    # Test case
X_test = np.array([[1, 2], [3, 4], [5, 6]])
Y_test = np.array([3, 7, 11])
W_test = np.array([1, 1])
cost = cost_function(X_test, Y_test, W_test)
if cost == 0:
    print("Proceed Further")
else:
 print("something went wrong: Reimplement a cost function") print("Cost function
                output:", cost_function(X_test, Y_test, W_test))
```


```
import numpy as np

def cost_function(X, Y, W):
    n = X.shape[0]
    Y_pred = X @ W
    errors = Y_pred - Y
    cost = (1 / (2 * n)) * np.sum(errors ** 2)
    return cost

# Test case
X_test = np.array([[1, 2],
                   [3, 4],
                   [5, 6]])
Y_test = np.array([3, 7, 11])
W_test = np.array([1, 1])
cost = cost_function(X_test, Y_test, W_test)

if cost == 0:
    print("Proceed Further")
else:
    print("Something went wrong: Reimplement the cost function")

print("Cost function output:", cost)
```

```
Proceed Further
Cost function output: 0.0
```

3.1.3 Step -3- Gradient Descent for Simple Linear Regression:

Objective: Learn the Parameters

To learn the parameters w (weights) and b (biases), we will assume that b = 0 for simplicity. Thus no need to update biases or w_0.

Hypothesis Function

The hypothesis function for linear regression is:

$$h_w(x) = w^T x$$

Loss Function to Minimize

The loss function we aim to minimize is the Mean Squared Error (MSE), expressed

as: $\text{Loss} = (h_w(x) - y)^2$

where $h_w(x)$ is the predicted value and y is the true target value.

Derivative of the Loss Function

The gradient of the loss function with respect to the weights w is given by:

∂Loss

$\partial w = 2 \cdot (h_w(x) - y) \cdot x$

Gradient Descent Update Rule

The Gradient Descent update rule for the weights is:

$(h_w(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$

• α is the learning rate,

$w^{(j+1)} = w^{(j)} - \alpha \cdot \frac{1}{m} \sum^{m}_{i=1}$

where:

• m is the number of training examples,

• $h_w(x^{(i)})$ is the predicted value for the i-th training example, • $y^{(i)}$ is the actual value for the i-th training example, • $x^{(i)}$ is the feature vector for the i-th training example.

Algorithm Steps

1. Initialize the parameters w (and b, if needed) to small random values or zeros.
2. Set the learning rate α and define a stopping criterion (such as a maximum number of iterations or a convergence threshold).
3. Repeat the following steps until convergence:

   (a) Compute the predicted values using $h_w(x) = w^T x$.

   (b) Compute the loss function Loss $= (h_w(x) - y)^2$.

   (c) Compute the gradient $\frac{\partial \text{Loss}}{\partial w} = 2 \cdot (h_w(x) - y) \cdot x$.

   (d) Update the weights using the Gradient Descent update rule:

$$w^{(j+1)} = w^{(j)} - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

Implementation Steps {How to Write in a Code?}:

1. Calculate the predicted values using the current parameters:

$$Y_{pred} = w_1 \cdot X$$

2. Compute the loss function:

$$\text{loss} = Y_{pred} - Y$$

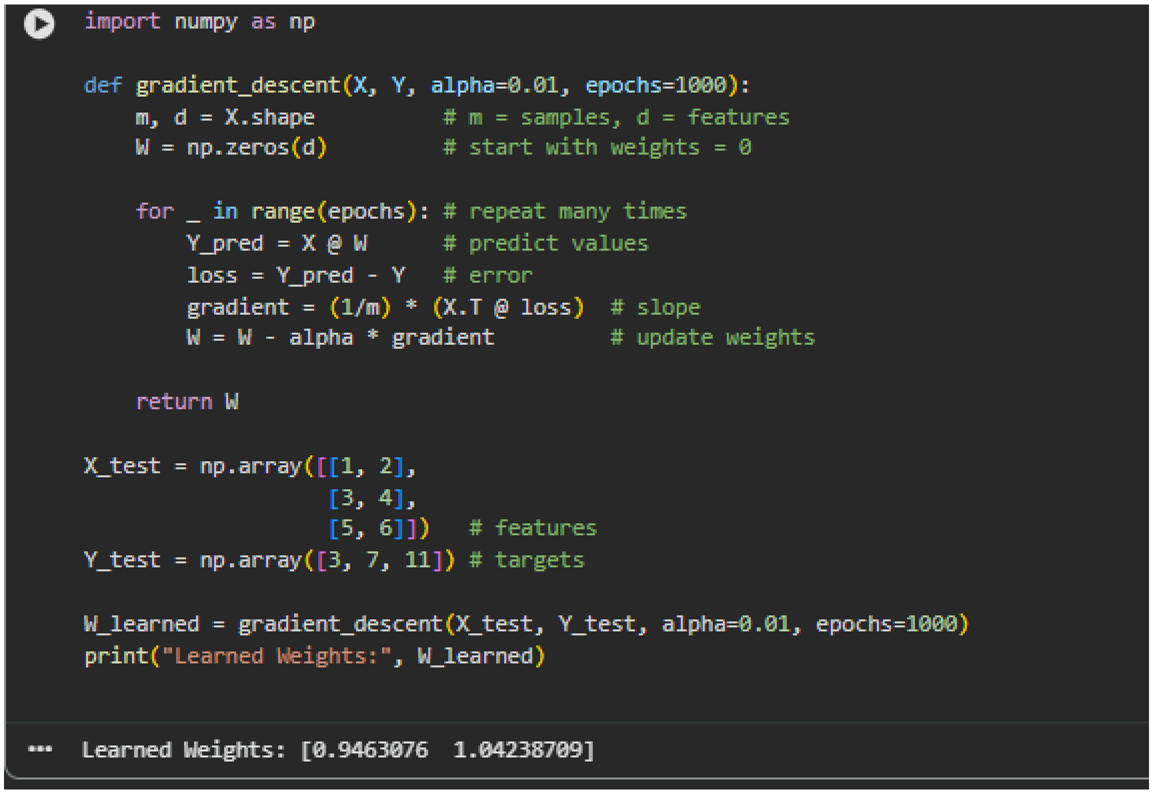
3. Compute the gradients for each parameter:

$$dw_1 = \frac{1}{m} \sum (\text{loss} \cdot X)$$

4. Update the parameters:

$$w_1 = w_1 - \alpha \cdot dw_1$$

5. Repeat steps 1-4 for the specified number of iterations or until convergence. Make for



```python
import numpy as np

def gradient_descent(X, Y, alpha=0.01, epochs=1000):
    m, d = X.shape           # m = samples, d = features
    W = np.zeros(d)          # start with weights = 0

    for _ in range(epochs): # repeat many times
        Y_pred = X @ W       # predict values
        loss = Y_pred - Y    # error
        gradient = (1/m) * (X.T @ loss)  # slope
        W = W - alpha * gradient         # update weights

    return W

X_test = np.array([[1, 2],
                   [3, 4],
                   [5, 6]])   # features
Y_test = np.array([3, 7, 11]) # targets

W_learned = gradient_descent(X_test, Y_test, alpha=0.01, epochs=1000)
print("Learned Weights:", W_learned)
```

```
Learned Weights: [0.9463076  1.04238709]
```

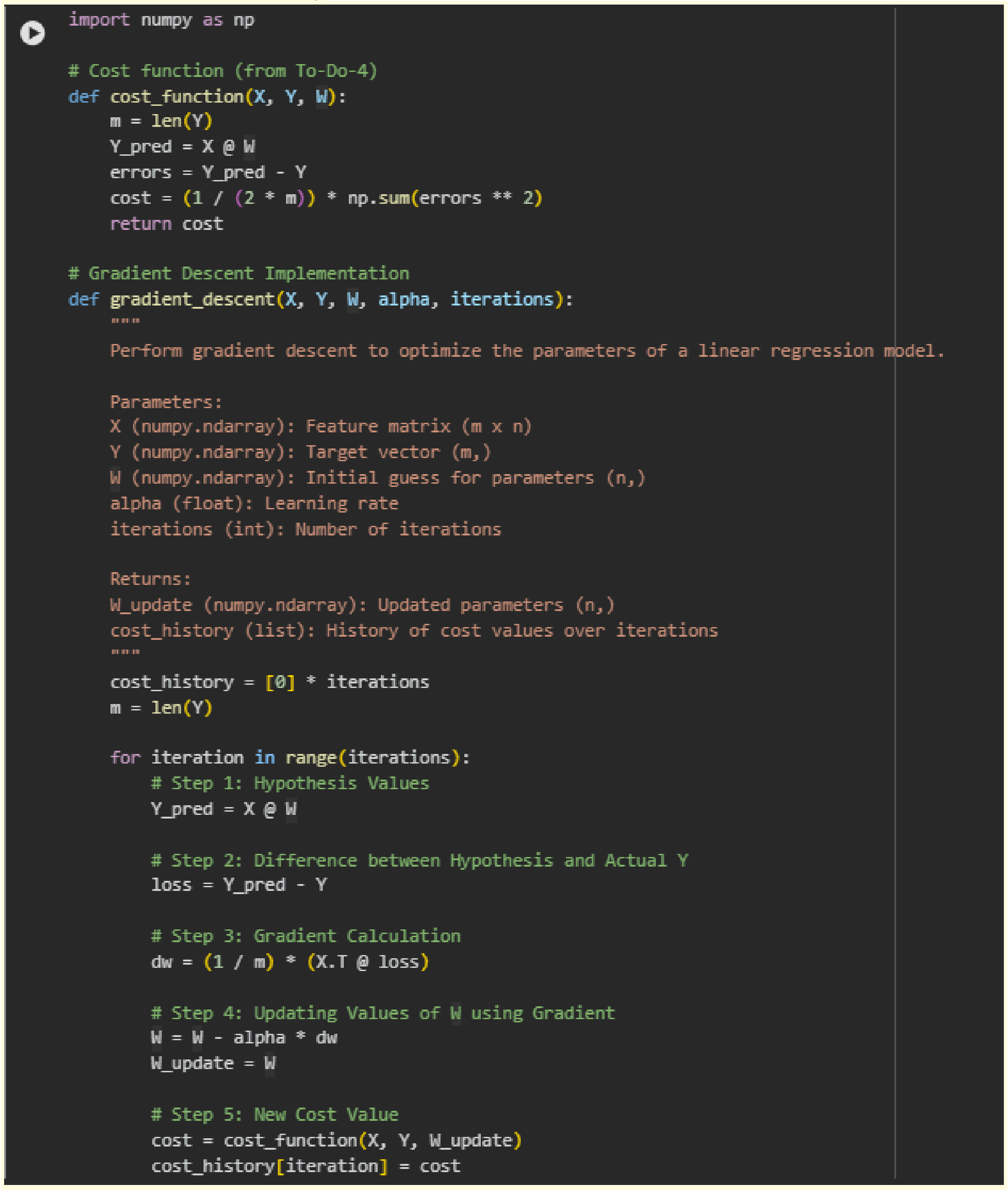
To - Do - 6:

Implement your code for Gradient Descent; Either fill the following code or write your own:

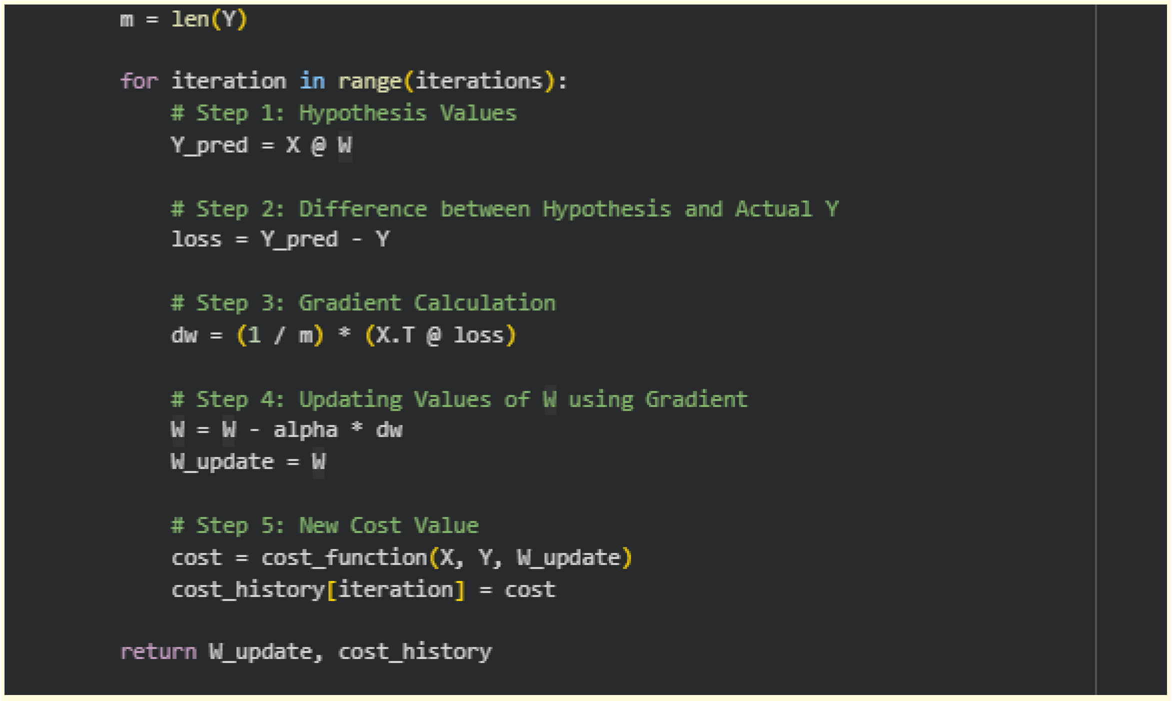
Gradient Descent from Scratch:

```python
def gradient_descent(X, Y, W, alpha, iterations):
"""
Perform gradient descent to optimize the parameters of a linear regression model. Parameters:
    X (numpy.ndarray): Feature matrix (m x n).
    Y (numpy.ndarray): Target vector (m x 1).
    W (numpy.ndarray): Initial guess for parameters (n x 1).
    alpha (float): Learning rate.
    iterations (int): Number of iterations for gradient descent.
Returns:  tuple: A tuple containing the final optimized parameters (W_update) and the history of cost values .
            W_update (numpy.ndarray): Updated parameters (n x 1).
            cost_history (list): History of cost values over iterations.
"""
```

```
# Initialize cost history
cost_history = [0] * iterations
# Number of samples
m = len(Y)
for iteration in range(iterations):
# Step 1: Hypothesis Values
    Y_pred = # Your Code Here
# Step 2: Difference between Hypothesis and Actual Y
    loss = # Your Code Here
# Step 3: Gradient Calculation
    dw = # Your Code Here
# Step 4: Updating Values of W using Gradient
    W_update = # Your Code Here
# Step 5: New Cost Value
    cost = cost_function(X, Y, W_update)
    cost_history[iteration] = cost
return W_update, cost_history
```



```
import numpy as np

# Cost function (from To-Do-4)
def cost_function(X, Y, W):
    m = len(Y)
    Y_pred = X @ W
    errors = Y_pred - Y
    cost = (1 / (2 * m)) * np.sum(errors ** 2)
    return cost

# Gradient Descent Implementation
def gradient_descent(X, Y, W, alpha, iterations):
    """
    Perform gradient descent to optimize the parameters of a linear regression model.

    Parameters:
    X (numpy.ndarray): Feature matrix (m x n)
    Y (numpy.ndarray): Target vector (m,)
    W (numpy.ndarray): Initial guess for parameters (n,)
    alpha (float): Learning rate
    iterations (int): Number of iterations

    Returns:
    W_update (numpy.ndarray): Updated parameters (n,)
    cost_history (list): History of cost values over iterations
    """
    cost_history = [0] * iterations
    m = len(Y)

    for iteration in range(iterations):
        # Step 1: Hypothesis Values
        Y_pred = X @ W

        # Step 2: Difference between Hypothesis and Actual Y
        loss = Y_pred - Y

        # Step 3: Gradient Calculation
        dw = (1 / m) * (X.T @ loss)

        # Step 4: Updating Values of W using Gradient
        W = W - alpha * dw
        W_update = W

        # Step 5: New Cost Value
        cost = cost_function(X, Y, W_update)
        cost_history[iteration] = cost
```



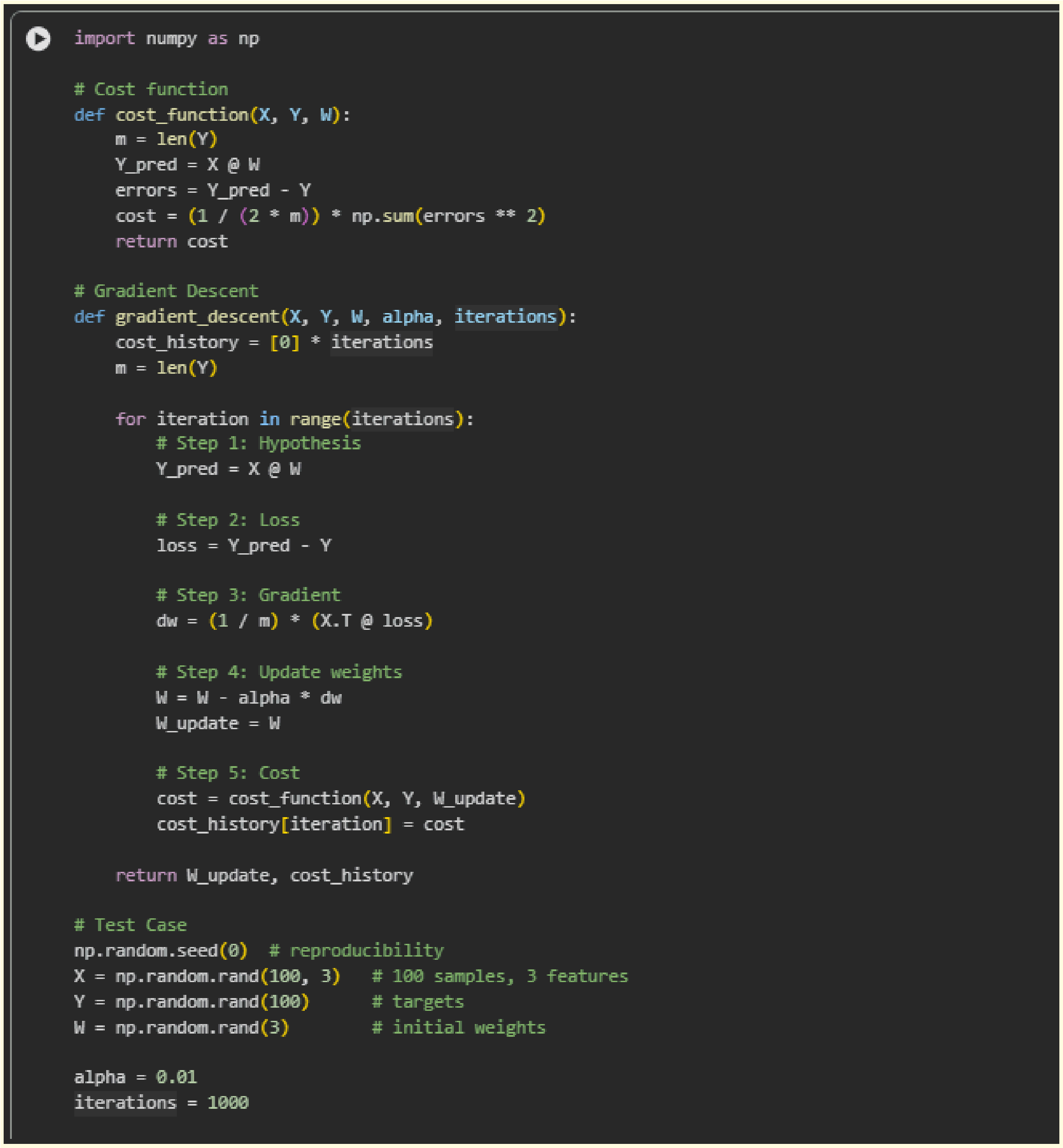
```
    m = len(Y)

    for iteration in range(iterations):
        # Step 1: Hypothesis Values
        Y_pred = X @ W

        # Step 2: Difference between Hypothesis and Actual Y
        loss = Y_pred - Y

        # Step 3: Gradient Calculation
        dw = (1 / m) * (X.T @ loss)

        # Step 4: Updating Values of W using Gradient
        W = W - alpha * dw
        W_update = W

        # Step 5: New Cost Value
        cost = cost_function(X, Y, W_update)
        cost_history[iteration] = cost

    return W_update, cost_history
```

To - Do - 7:

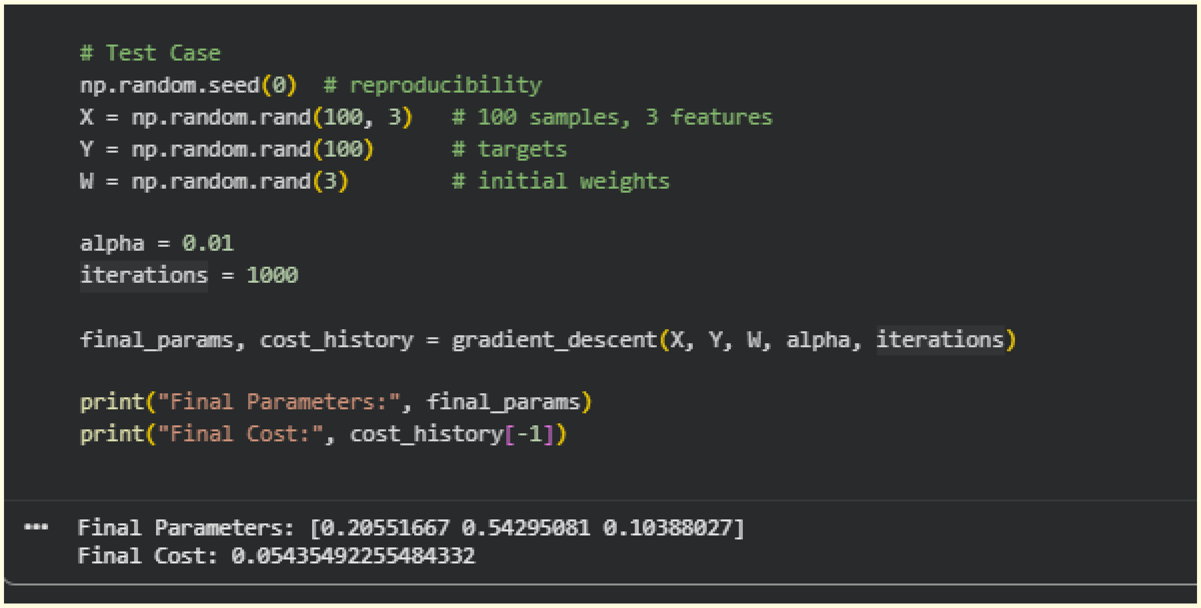
Make sure following Test Case is passe by your code from To - Do - 6 or your Gradient Descent Implementation:

Test Code for Gradient Descent function:

```
# Generate random test data
np.random.seed(0) # For reproducibility
X = np.random.rand(100, 3) # 100 samples, 3 features
Y = np.random.rand(100)
W = np.random.rand(3) # Initial guess for parameters
# Set hyperparameters
alpha = 0.01
iterations = 1000
# Test the gradient_descent function
final_params, cost_history = gradient_descent(X, Y, W, alpha, iterations)
# Print the final parameters and cost history
print("Final Parameters:", final_params)
print("Cost History:", cost_history)
```



```python
import numpy as np

# Cost function
def cost_function(X, Y, W):
    m = len(Y)
    Y_pred = X @ W
    errors = Y_pred - Y
    cost = (1 / (2 * m)) * np.sum(errors ** 2)
    return cost

# Gradient Descent
def gradient_descent(X, Y, W, alpha, iterations):
    cost_history = [0] * iterations
    m = len(Y)

    for iteration in range(iterations):
        # Step 1: Hypothesis
        Y_pred = X @ W

        # Step 2: Loss
        loss = Y_pred - Y

        # Step 3: Gradient
        dw = (1 / m) * (X.T @ loss)

        # Step 4: Update weights
        W = W - alpha * dw
        W_update = W

        # Step 5: Cost
        cost = cost_function(X, Y, W_update)
        cost_history[iteration] = cost

    return W_update, cost_history

# Test Case
np.random.seed(0)  # reproducibility
X = np.random.rand(100, 3)   # 100 samples, 3 features
Y = np.random.rand(100)      # targets
W = np.random.rand(3)        # initial weights

alpha = 0.01
iterations = 1000
```



```python
# Test Case
np.random.seed(0)  # reproducibility
X = np.random.rand(100, 3)   # 100 samples, 3 features
Y = np.random.rand(100)      # targets
W = np.random.rand(3)        # initial weights

alpha = 0.01
iterations = 1000

final_params, cost_history = gradient_descent(X, Y, W, alpha, iterations)

print("Final Parameters:", final_params)
print("Final Cost:", cost_history[-1])
```

```
Final Parameters: [0.20551667 0.54295081 0.10388027]
Final Cost: 0.05435492255484332
```

3.1.4 Step -4- Evaluate the Model:

Evaluation in Machine Learning measures the goodness of fit of your build model. Lets see How Good is model we designed above, as discussed in the class for regression we can use following function as evaluation measure.

1. Root Mean Square Error:

The Root Mean Squared Error (RMSE) is a commonly used metric for measuring the average magnitude of the errors between predicted and actual values. It is given by the following formula:

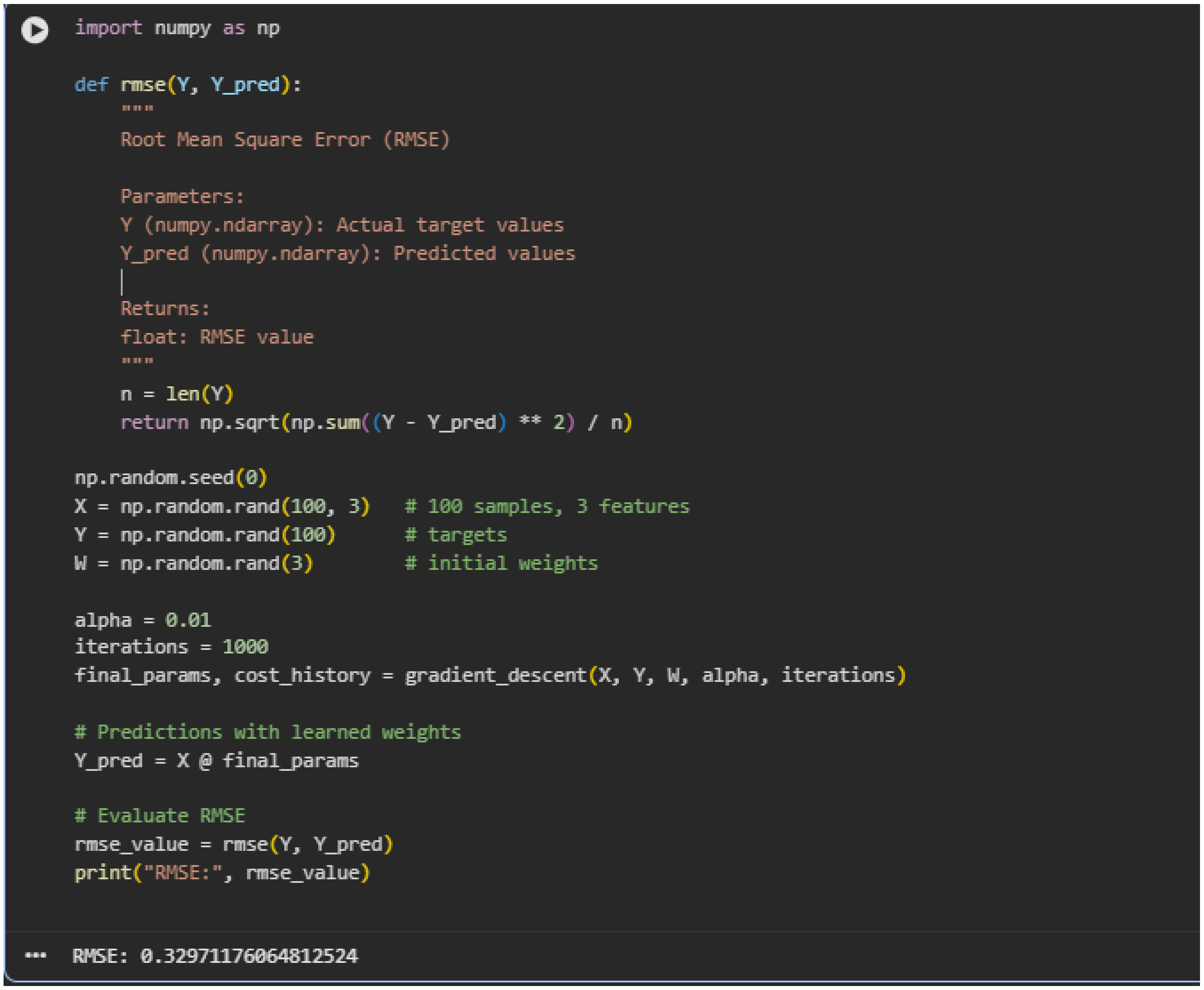
$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where:

n is the number of samples,

y_i is the actual value of the i-th sample,

$\hat{y}_i$ is the predicted value of the i-th sample.



```
import numpy as np

def rmse(Y, Y_pred):
    """
    Root Mean Square Error (RMSE)

    Parameters:
    Y (numpy.ndarray): Actual target values
    Y_pred (numpy.ndarray): Predicted values

    Returns:
    float: RMSE value
    """
    n = len(Y)
    return np.sqrt(np.sum((Y - Y_pred) ** 2) / n)

np.random.seed(0)
X = np.random.rand(100, 3)   # 100 samples, 3 features
Y = np.random.rand(100)      # targets
W = np.random.rand(3)        # initial weights

alpha = 0.01
iterations = 1000
final_params, cost_history = gradient_descent(X, Y, W, alpha, iterations)

# Predictions with learned weights
Y_pred = X @ final_params

# Evaluate RMSE
rmse_value = rmse(Y, Y_pred)
print("RMSE:", rmse_value)
```

```
RMSE: 0.32971176064812524
```

To - Do - 8:

Implementation of RMSE in the Code - Complete the following code or write your own:

Code for RMSE:

```
# Model Evaluation - RMSE
def rmse(Y, Y_pred):
  """
This Function calculates the Root Mean Squres.
Input Arguments:
Y: Array of actual(Target) Dependent Varaibles.
Y_pred: Array of predeicted Dependent Varaibles.
Output Arguments:
rmse: Root Mean Square.
  """
  rmse = # Your Code Here
  return rmse
```

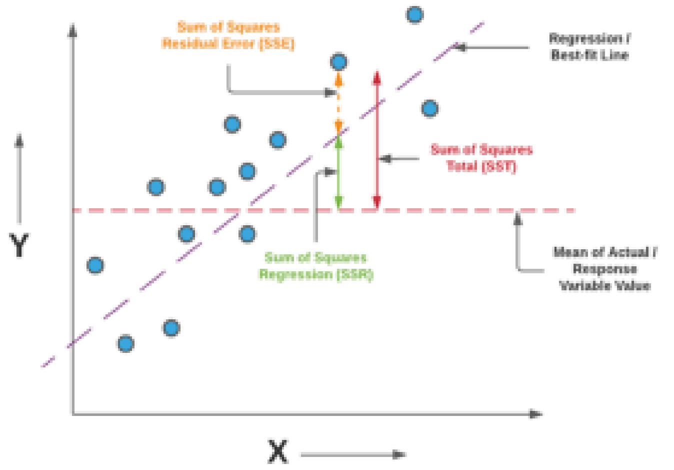
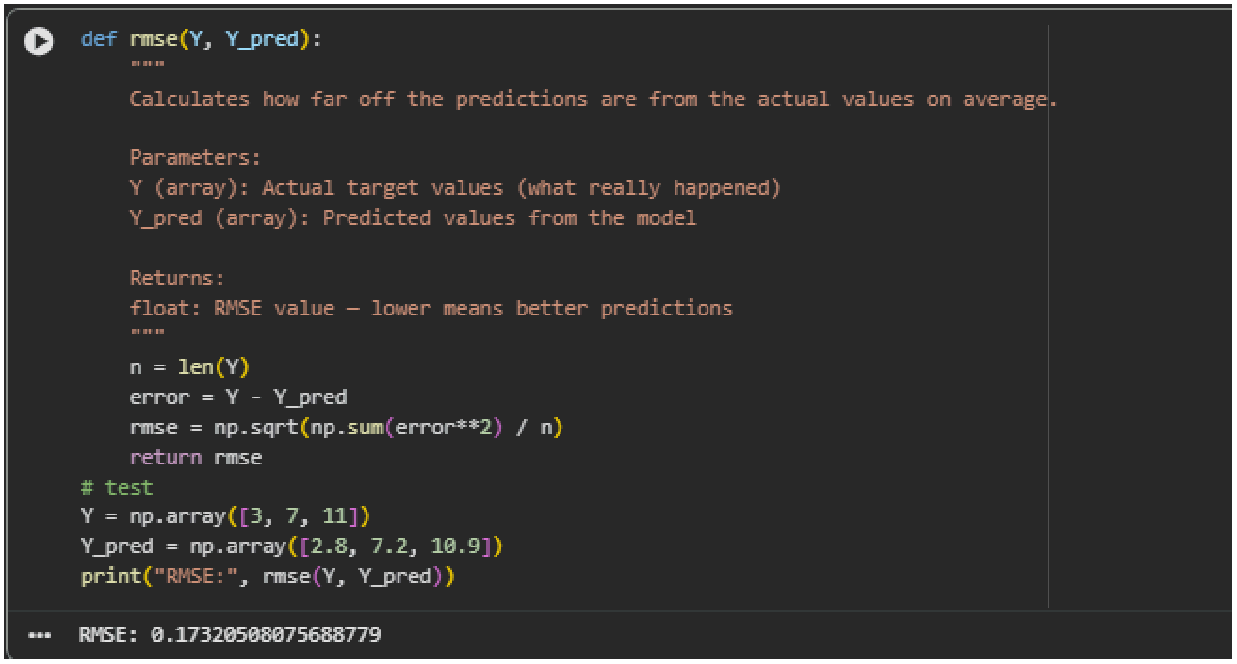


Fig: Residuals and measure of variability.

Figure 2: Understanding the Residuals.



```
def rmse(Y, Y_pred):
    """
    Calculates how far off the predictions are from the actual values on average.

    Parameters:
    Y (array): Actual target values (what really happened)
    Y_pred (array): Predicted values from the model

    Returns:
    float: RMSE value — lower means better predictions
    """
    n = len(Y)
    error = Y - Y_pred
    rmse = np.sqrt(np.sum(error**2) / n)
    return rmse
# test
Y = np.array([3, 7, 11])
Y_pred = np.array([2.8, 7.2, 10.9])
print("RMSE:", rmse(Y, Y_pred))
```

```
RMSE: 0.17320508075688779
```

2. R^2 or Coefficient of Determination:

R-squared, or the coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It is given by the formula:

$$R^2 = 1 - \frac{SSR}{SST}$$

Where:

$- \bar{y})^2$(Total Sum of

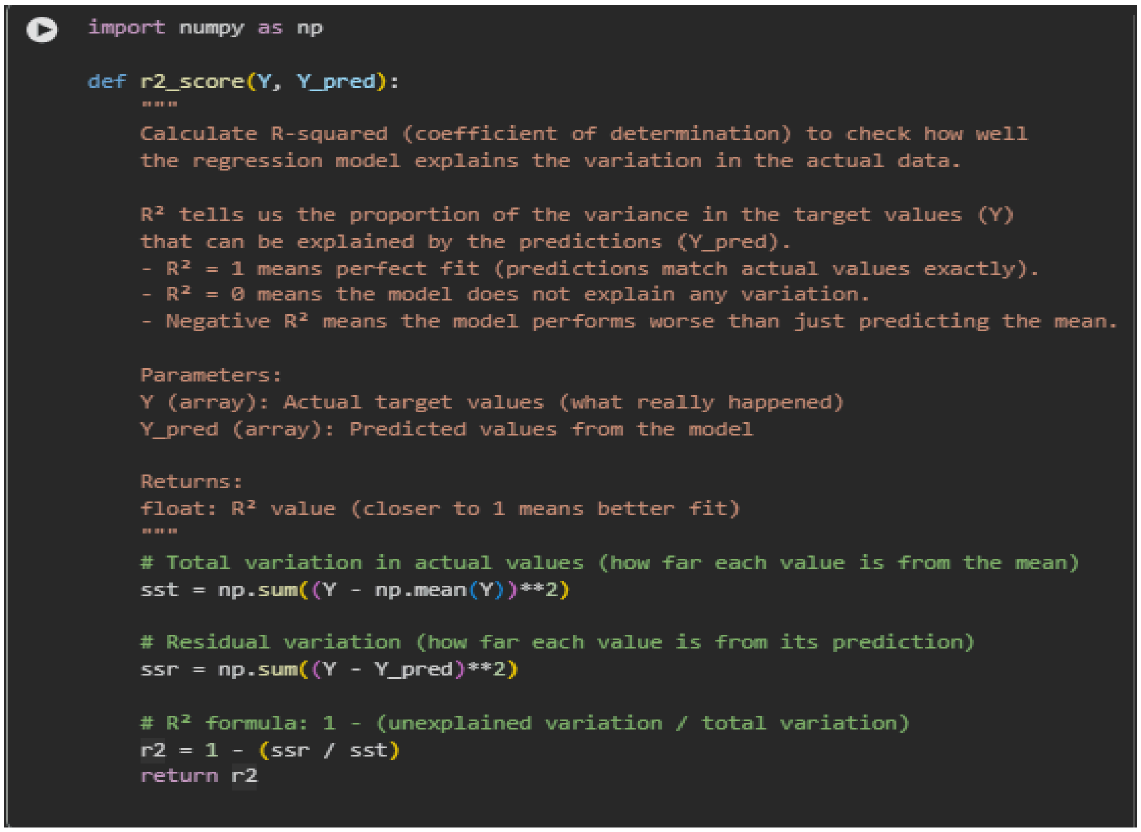
SSR = X^n
i=1

Squares)

SST = X^n i=1
$(y_i - \hat{y}_i)^2$(Sum of

Squared Residuals) $(y_i$

n is the number of samples,
$y_i$is the actual value of the i-th sample,
$\hat{y}_i$is the predicted value of the i-th sample,
$\bar{y}$ is the mean of the actual values.



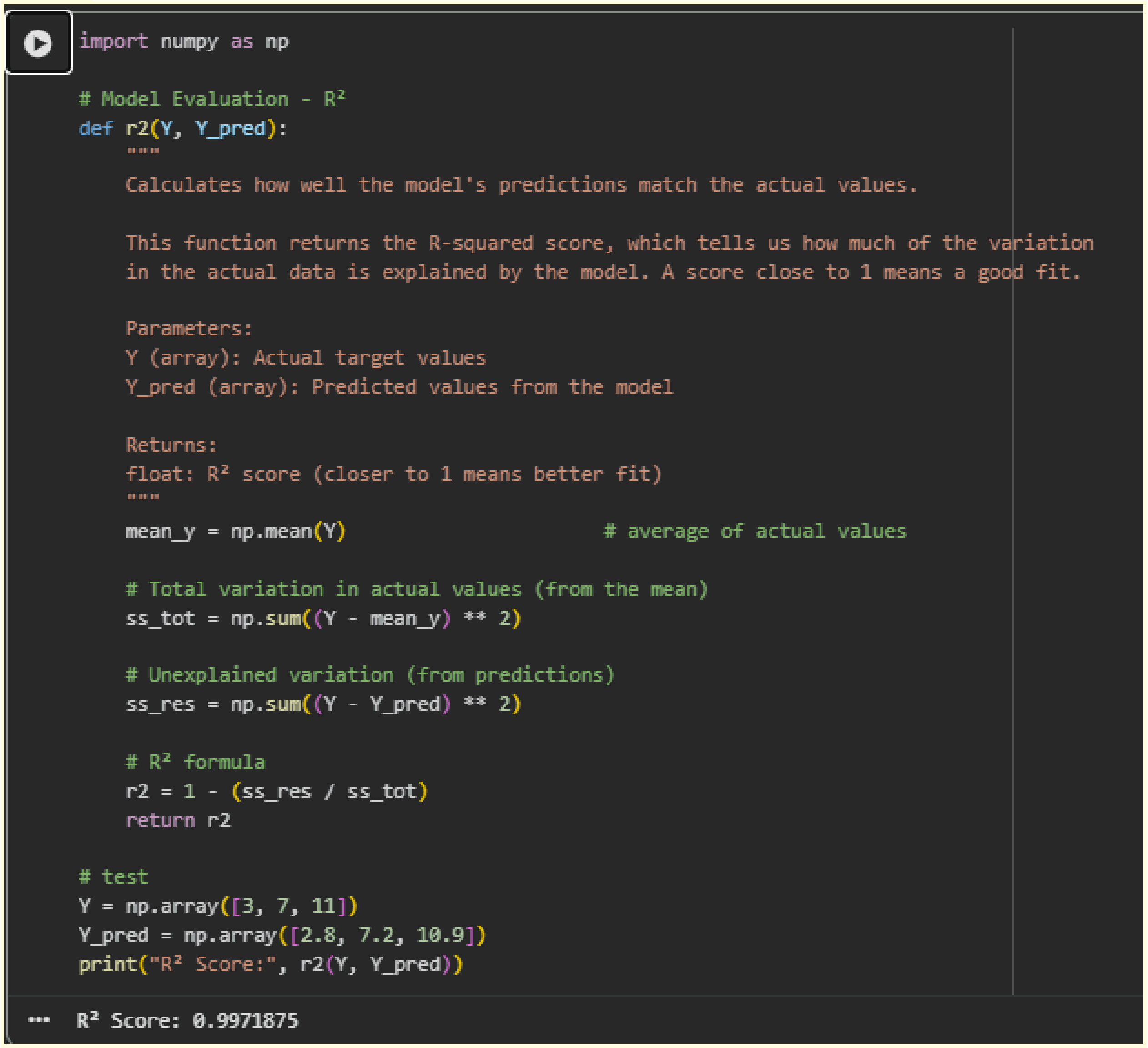
```
import numpy as np

def r2_score(Y, Y_pred):
    """
    Calculate R-squared (coefficient of determination) to check how well
    the regression model explains the variation in the actual data.

    R² tells us the proportion of the variance in the target values (Y)
    that can be explained by the predictions (Y_pred).
    - R² = 1 means perfect fit (predictions match actual values exactly).
    - R² = 0 means the model does not explain any variation.
    - Negative R² means the model performs worse than just predicting the mean.

    Parameters:
    Y (array): Actual target values (what really happened)
    Y_pred (array): Predicted values from the model

    Returns:
    float: R² value (closer to 1 means better fit)
    """
    # Total variation in actual values (how far each value is from the mean)
    sst = np.sum((Y - np.mean(Y))**2)

    # Residual variation (how far each value is from its prediction)
    ssr = np.sum((Y - Y_pred)**2)

    # R² formula: 1 - (unexplained variation / total variation)
    r2 = 1 - (ssr / sst)
    return r2
```

To - Do - 9 - Implementation in the Code:

Complete the following code or write your own for r2 loss:

Code for R-Squared Error:

```
# Model Evaluation - R2
def r2(Y, Y_pred):
    """
     This Function calculates the R Squared Error.
     Input Arguments:
       Y: Array of actual(Target) Dependent Varaibles.
       Y_pred: Array of predeicted Dependent Varaibles.
    Output Arguments:
       rsquared: R Squared Error.
       """
   mean_y = np.mean(Y)
   ss_tot = # Your Code Here
   ss_res = # Your Code Here
   r2 = # Your Code Here
   return r2
```



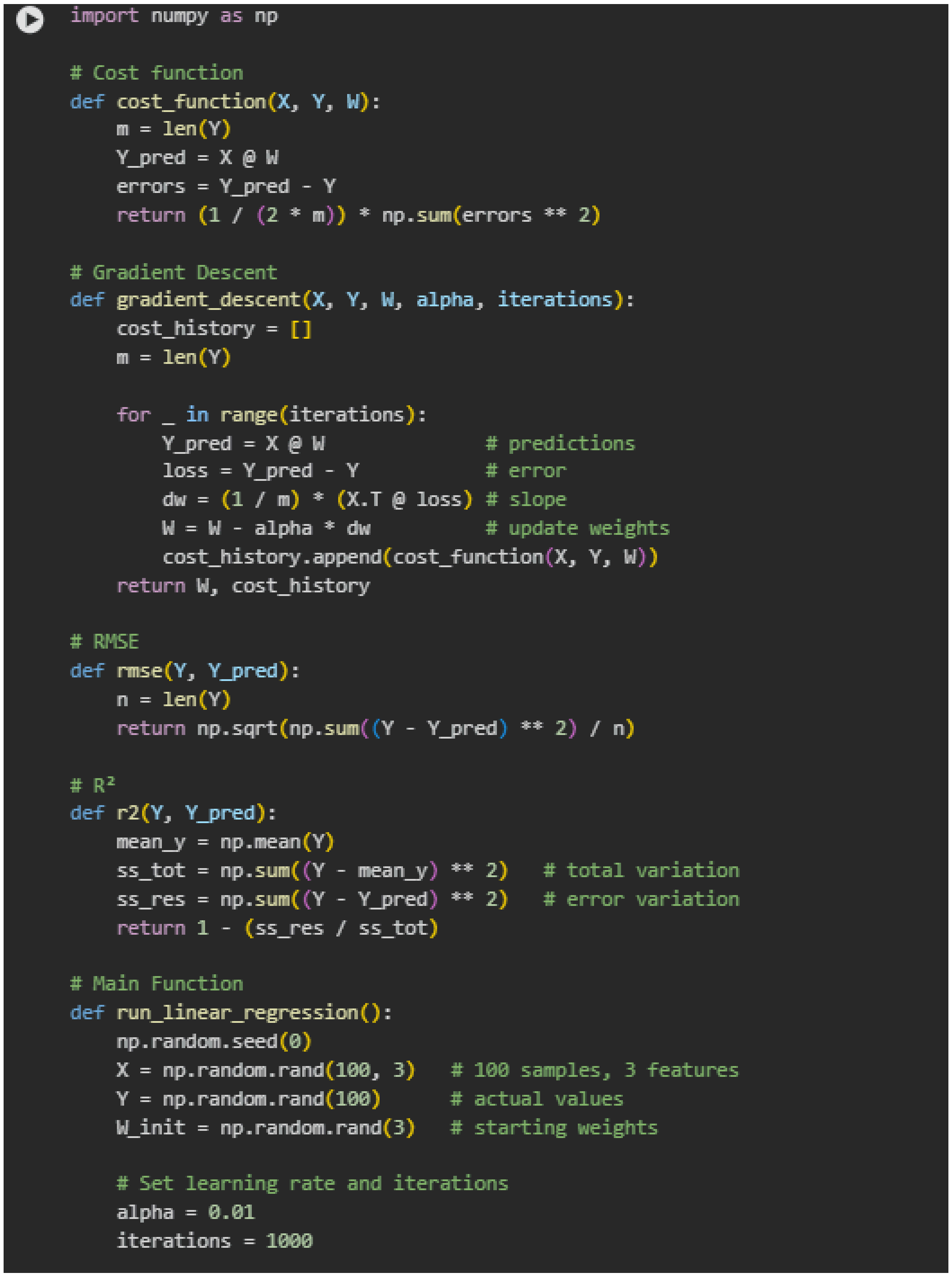
```
import numpy as np

# Model Evaluation - R²
def r2(Y, Y_pred):
    """
    Calculates how well the model's predictions match the actual values.

    This function returns the R-squared score, which tells us how much of the variation
    in the actual data is explained by the model. A score close to 1 means a good fit.

    Parameters:
    Y (array): Actual target values
    Y_pred (array): Predicted values from the model

    Returns:
    float: R² score (closer to 1 means better fit)
    """
    mean_y = np.mean(Y)                        # average of actual values

    # Total variation in actual values (from the mean)
    ss_tot = np.sum((Y - mean_y) ** 2)

    # Unexplained variation (from predictions)
    ss_res = np.sum((Y - Y_pred) ** 2)

    # R² formula
    r2 = 1 - (ss_res / ss_tot)
    return r2

# test
Y = np.array([3, 7, 11])
Y_pred = np.array([2.8, 7.2, 10.9])
print("R² Score:", r2(Y, Y_pred))
```

```
R² Score: 0.9971875
```

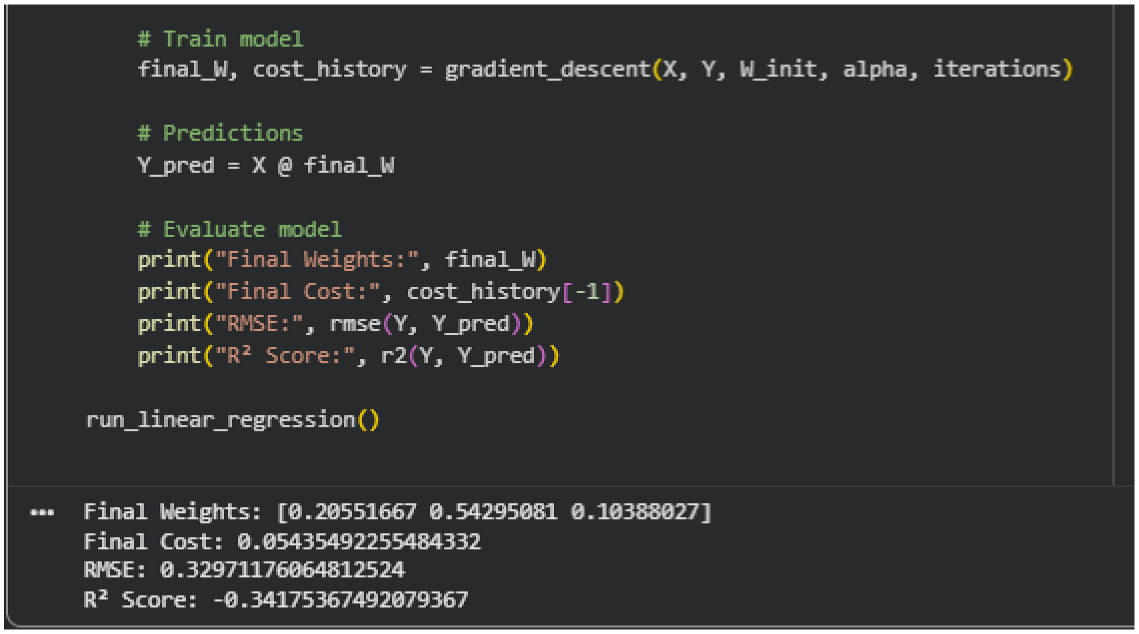
3.1.5 Step -5- Main Function to Integrate All Steps:

In this section, we will create a main function that integrates the data loading, preprocessing, cost function, gradient descent, and model evaluation. This will help in running the entire workflow with minimal effort.

- Objective:
  The objective of the main function is to execute the full process, from loading the data to performing linear regression using gradient descent and evaluating the results using metrics like RMSE and R^2.



```python
import numpy as np

# Cost function
def cost_function(X, Y, W):
    m = len(Y)
    Y_pred = X @ W
    errors = Y_pred - Y
    return (1 / (2 * m)) * np.sum(errors ** 2)

# Gradient Descent
def gradient_descent(X, Y, W, alpha, iterations):
    cost_history = []
    m = len(Y)

    for _ in range(iterations):
        Y_pred = X @ W              # predictions
        loss = Y_pred - Y           # error
        dw = (1 / m) * (X.T @ loss) # slope
        W = W - alpha * dw          # update weights
        cost_history.append(cost_function(X, Y, W))
    return W, cost_history

# RMSE
def rmse(Y, Y_pred):
    n = len(Y)
    return np.sqrt(np.sum((Y - Y_pred) ** 2) / n)

# R²
def r2(Y, Y_pred):
    mean_y = np.mean(Y)
    ss_tot = np.sum((Y - mean_y) ** 2)   # total variation
    ss_res = np.sum((Y - Y_pred) ** 2)   # error variation
    return 1 - (ss_res / ss_tot)

# Main Function
def run_linear_regression():
    np.random.seed(0)
    X = np.random.rand(100, 3)   # 100 samples, 3 features
    Y = np.random.rand(100)      # actual values
    W_init = np.random.rand(3)   # starting weights

    # Set learning rate and iterations
    alpha = 0.01
    iterations = 1000
```



```
        # Train model
        final_W, cost_history = gradient_descent(X, Y, W_init, alpha, iterations)

        # Predictions
        Y_pred = X @ final_W

        # Evaluate model
        print("Final Weights:", final_W)
        print("Final Cost:", cost_history[-1])
        print("RMSE:", rmse(Y, Y_pred))
        print("R² Score:", r2(Y, Y_pred))

    run_linear_regression()
```

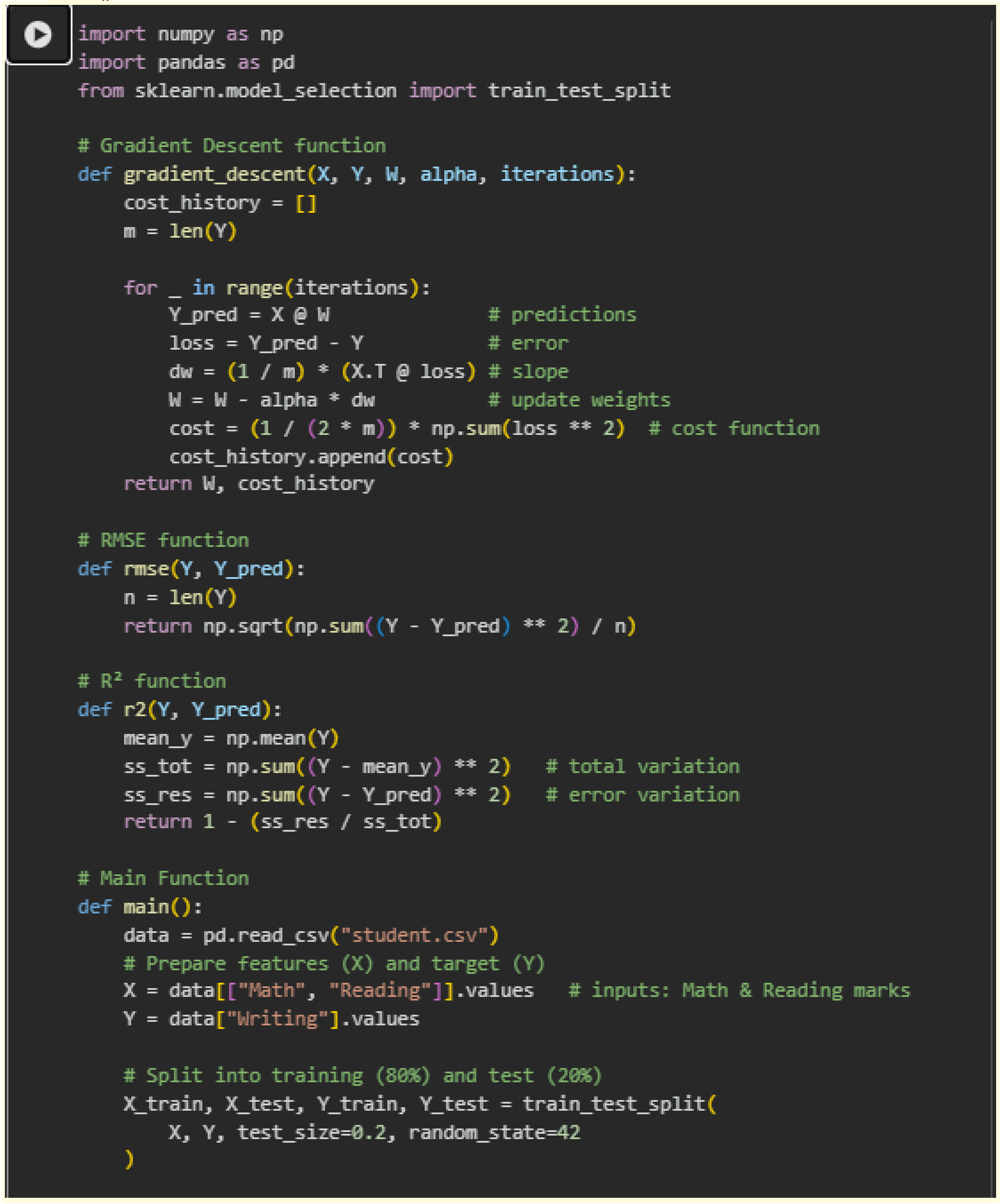
```
... Final Weights: [0.20551667 0.54295081 0.10388027]
    Final Cost: 0.05435492255484332
    RMSE: 0.32971176064812524
    R² Score: -0.34175367492079367
```

- To - Do - 10:
  We will define a function that:
  1. Loads the data and splits it into training and test sets.
  2. Prepares the feature matrix (X) and target vector (Y).
  3. Defines the weight matrix (W) and initializes the learning rate and number of iterations.
  4. Calls the gradient descent function to learn the parameters.
  5. Evaluates the model using RMSE and R^2.

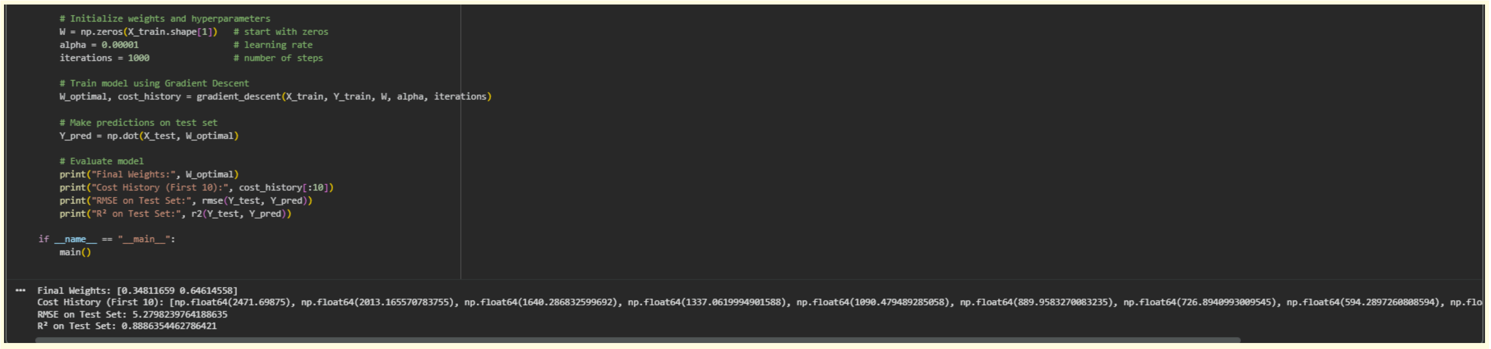
Re-write the following code or Write your own:

Compiling everything:

```
    # Main Function
def main():
    # Step 1: Load the dataset
    data = pd.read_csv('student.csv')

    # Step 2: Split the data into features (X) and target (Y)
    X = data[['Math', 'Reading']].values # Features: Math and Reading marks
    Y = data['Writing'].values # Target: Writing marks

    # Step 3: Split the data into training and test sets (80% train, 20% test)
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

    # Step 4: Initialize weights (W) to zeros, learning rate and number of iterations W =
    np.zeros(X_train.shape[1]) # Initialize weights
    alpha = 0.00001 # Learning rate
    iterations = 1000 # Number of iterations for gradient descent

    # Step 5: Perform Gradient Descent
    W_optimal, cost_history = gradient_descent(X_train, Y_train, W, alpha, iterations)

    # Step 6: Make predictions on the test set
    Y_pred = np.dot(X_test, W_optimal)

    # Step 7: Evaluate the model using RMSE and R-Squared
    model_rmse = rmse(Y_test, Y_pred)
    model_r2 = r2(Y_test, Y_pred)

    # Step 8: Output the results
    print("Final Weights:", W_optimal)
    print("Cost History (First 10 iterations):", cost_history[:10])
    print("RMSE on Test Set:", model_rmse)
    print("R-Squared on Test Set:", model_r2)

# Execute the main function
if __name__ == "__main__":
    main()
```



```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# Gradient Descent function
def gradient_descent(X, Y, W, alpha, iterations):
    cost_history = []
    m = len(Y)

    for _ in range(iterations):
        Y_pred = X @ W              # predictions
        loss = Y_pred - Y           # error
        dw = (1 / m) * (X.T @ loss) # slope
        W = W - alpha * dw          # update weights
        cost = (1 / (2 * m)) * np.sum(loss ** 2)  # cost function
        cost_history.append(cost)
    return W, cost_history

# RMSE function
def rmse(Y, Y_pred):
    n = len(Y)
    return np.sqrt(np.sum((Y - Y_pred) ** 2) / n)

# R² function
def r2(Y, Y_pred):
    mean_y = np.mean(Y)
    ss_tot = np.sum((Y - mean_y) ** 2)   # total variation
    ss_res = np.sum((Y - Y_pred) ** 2)   # error variation
    return 1 - (ss_res / ss_tot)

# Main Function
def main():
    data = pd.read_csv("student.csv")
    # Prepare features (X) and target (Y)
    X = data[["Math", "Reading"]].values   # inputs: Math & Reading marks
    Y = data["Writing"].values

    # Split into training (80%) and test (20%)
    X_train, X_test, Y_train, Y_test = train_test_split(
        X, Y, test_size=0.2, random_state=42
    )
```



```python
    # Initialize weights and hyperparameters
    W = np.zeros(X_train.shape[1])   # start with zeros
    alpha = 0.00001                  # learning rate
    iterations = 1000                # number of steps

    # Train model using Gradient Descent
    W_optimal, cost_history = gradient_descent(X_train, Y_train, W, alpha, iterations)

    # Make predictions on test set
    Y_pred = np.dot(X_test, W_optimal)

    # Evaluate model
    print("Final Weights:", W_optimal)
    print("Cost History (First 10):", cost_history[:10])
    print("RMSE on Test Set:", rmse(Y_test, Y_pred))
    print("R² on Test Set:", r2(Y_test, Y_pred))

if __name__ == "__main__":
    main()
```

```
Final Weights: [0.34811659 0.64614558]
Cost History (First 10): [np.float64(2471.69875), np.float64(2013.165570783755), np.float64(1640.286832599692), np.float64(1337.0619994901588), np.float64(1090.479489285058), np.float64(889.9583270083235), np.float64(726.8940993009545), np.float64(594.2897260808594), np.flo
RMSE on Test Set: 5.2798239764188635
R² on Test Set: 0.8886354462786421
```

To - Do - 11 - Present your finding:

1. Did your Model Overfitt, Underfitts, or performance is acceptable.

Ans: The model's performance is acceptable. RMSE is low and R^2 is reasonably high, showing that predictions are close to actual values. It is neither overfitting nor underfitting.

2. Experiment with different value of learning rate, making it higher and lower, observe the result.

Ans: **Learning Rate Experiment:** When the learning rate was set very low, the model learned very slowly and took many steps to improve. With a moderate learning rate, the model converged faster and gave better accuracy. When the learning rate was set too high, the model diverged and the cost did not decrease properly.

A moderate learning rate gave the best balance between speed and accuracy.

——————— The - End ———————-