• **To - Do - 1:**

1. Read and Observe the Dataset.
2. Print top(5) and bottom(5) of the dataset {Hint: pd.head and pd.tail}.
3. Print the Information of Datasets. {Hint: pd.info}.
4. Gather the Descriptive info about the Dataset. {Hint: pd.describe}
5. Split your data into Feature (X) and Label (Y).

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

```
# Step 1: Read and observe the dataset
data = pd.read_csv('/content/drive/MyDrive/student.csv')
```

```
# Step 2: Print the top 5 and bottom 5 rows of the dataset
print("Top 5 rows of the dataset:")
print(data.head())

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

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

```
# Step 3: Print the information of the dataset
print("\nDataset Information:")
print(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
None
```

```
# Step 4: Gather descriptive info about the dataset
print("\nDescriptive 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
```

```
# Step 5: Split data into Feature (X) and Label (Y)
X = data[['Math', 'Reading']].values  # Features: Math and Reading marks
Y = data['Writing'].values  # Target: Writing marks

# Print out the shapes of X and Y to verify
print("\nShape of Features (X):", X.shape)
print("Shape of Target (Y):", Y.shape)
```

```

Shape of Features (X): (1000, 2)
Shape of Target (Y): (1000,)
```

To - Do - 2:

```
# Step 1: Extract the feature matrix X and the target vector Y
# Assume the columns 'Math' and 'Reading' are the features, and 'Writing' is the target
X = data[['Math', 'Reading']].values  # Feature matrix (d x n)
Y = data['Writing'].values            # Target vector (n,)

# Step 2: Transpose X to match the dimension (d x n) -> (n x d) for matrix multiplication
X = X.T  # Now X is of shape (2, n), where 2 is the number of features

# Step 3: Initialize weights W (d x 1), with zeros (for simplicity)
W = np.zeros((X.shape[0], 1))  # W is of shape (d, 1), i.e., weights for each feature

# Step 4: Make predictions Y_pred using Y = W^T * X (dot product)
Y_pred = np.dot(W.T, X)  # Y_pred will have shape (1, n), and we want (n,)

# Reshape Y_pred to be a column vector for ease
Y_pred = Y_pred.T

# Step 5: Compute the Mean Squared Error (MSE) loss
mse_loss = np.mean((Y - Y_pred) ** 2)
print("Mean Squared Error (MSE):", mse_loss)

# Print the initial weight vector
print("Initial Weights (W):")
print(W)
```

```
Mean Squared Error (MSE): 4940.22
Initial Weights (W):
[[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.

```
# Step 1: Split the dataset into training and testing sets (80-20 split)
X_train, X_test, Y_train, Y_test = train_test_split(X.T, Y, test_size=0.2, random_state=42) # Transpose X back to (1000, 2)

# Check the dimensions of the split data
print("Training Data (X_train) Shape:", X_train.shape)
print("Testing Data (X_test) Shape:", X_test.shape)
print("Training Labels (Y_train) Shape:", Y_train.shape)
print("Testing Labels (Y_test) Shape:", Y_test.shape)

# Optionally, if you want to confirm the split visually:
# Print the first few entries of X_train and Y_train
print("\nFirst few entries of X_train:\n", X_train[:5])
print("\nFirst few entries of Y_train:\n", Y_train[:5])
```

```
Training Data (X_train) Shape: (800, 2)
Testing Data (X_test) Shape: (200, 2)
Training Labels (Y_train) Shape: (800,)
Testing Labels (Y_test) Shape: (200,)

First few entries of X_train:
 [[64 82]
 [62 70]
 [36 21]
 [81 70]
 [82 86]]

First few entries of Y_train:
```

```
[78 67 25 71 87]
```

3.1.2 Step -2- Build a Cost Function: Cost function is the average of loss function measured across the data point.

```
def cost_function(X, Y, W):
    """
    Parameters:
    X : numpy.ndarray
        Feature matrix (d x n), where d is the number of features and n is the number of samples.
    Y : numpy.ndarray
        Target vector (n, ), where n is the number of samples.
    W : numpy.ndarray
        Weight vector (d, ), where d is the number of features.

    Output:
    cost : float
        The accumulated mean squared error (MSE).
    """
    # Step 1: Calculate the predicted values (Y_hat) using the linear model
    Y_pred = np.dot(X, W)  # X * W gives the predicted values

    # Step 2: Calculate the squared errors between the predicted and actual values
    errors = Y - Y_pred

    # Step 3: Compute the Mean Squared Error (MSE)
    cost = np.mean(errors ** 2)  # MSE = average of squared errors

    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.

```
# Define the cost function
def cost_function(X, Y, W):
    """
    Parameters:
    X : numpy.ndarray
        Feature matrix (d x n), where d is the number of features and n is the number of samples.
    Y : numpy.ndarray
        Target vector (n, ), where n is the number of samples.
    W : numpy.ndarray
        Weight vector (d, ), where d is the number of features.

    Output:
    cost : float
        The accumulated mean squared error (MSE).
    """
    # Step 1: Calculate the predicted values (Y_hat) using the linear model
    Y_pred = np.dot(X, W)  # X * W gives the predicted values

    # Step 2: Calculate the squared errors between the predicted and actual values
    errors = Y - Y_pred

    # Step 3: Compute the Mean Squared Error (MSE)
    cost = np.mean(errors ** 2)  # MSE = average of squared errors

    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])

# Calculate the cost
cost = cost_function(X_test, Y_test, W_test)

# Check if the cost is as expected (0)
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
```

Gradient Descent from Scratch:

```
# Define the cost function
def cost_function(X, Y, W):
    """
    Parameters:
    X : numpy.ndarray
        Feature matrix (m x n), where m is the number of samples and n is the number of features.
    Y : numpy.ndarray
        Target vector (m, ), where m is the number of samples.
    W : numpy.ndarray
        Weight vector (n, ), where n is the number of features.

    Output:
    cost : float
        The accumulated mean squared error (MSE).
    """
    # Calculate the predicted values
    Y_pred = np.dot(X, W)

    # Calculate the squared errors
    errors = Y - Y_pred

    # Compute the Mean Squared Error (MSE)
    cost = np.mean(errors ** 2)

    return cost

# Define the gradient descent function
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:
    W_update : numpy.ndarray
        Updated parameters (n x 1).
    cost_history : list
        History of cost values over iterations.
    """
    # Initialize cost history
    cost_history = []

    # Number of samples (m)
    m = len(Y)

    # Gradient descent loop
    for iteration in range(iterations):
        # Step 1: Hypothesis Values (hθ(X) = X * W)
        Y_pred = np.dot(X, W)

        # Step 2: Loss (Difference between predicted and actual values)
        loss = Y_pred - Y

        # Step 3: Gradient Calculation (dw = (2/m) * X^T * loss)
        dw = (2/m) * np.dot(X.T, loss)

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

        # Step 5: Calculate the new cost value and store it in cost_history
        cost = cost_function(X, Y, W_update)
        cost_history.append(cost)
        # Update weights for the next iteration
        W = W_update

    return W_update, cost_history

# Example Usage:

# Sample dataset (Feature matrix X and target vector Y)
X = np.array([[1, 3, 5],
              [2, 4, 6],
              [1, 3, 5],
              [2, 4, 6]])  # (4 samples, 3 features)

Y = np.array([3, 7, 3, 7])  # Target values (4 samples)

# Initialize weights (W) randomly or set to zeros
W_init = np.zeros(X.shape[1])  # (3 features)

# Set learning rate and number of iterations
alpha = 0.01
iterations = 1000

# Call gradient descent
W_optimal, cost_history = gradient_descent(X, Y, W_init, alpha, iterations)

# Print results
print("Optimized Weights:", W_optimal)
print("Final Cost:", cost_history[-1])
print("Cost History (first 10 iterations):", cost_history[:10])
```

```
Optimized Weights: [ 3.34235416  1.26765577 -0.80704262]
Final Cost: 0.009839845695018308
Cost History (first 10 iterations): [2.1872499999999997, 1.9468728849999999, 1.9346233499604994, 1.9243856735727318, 1.9
```

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)
```

```
Final Parameters: [0.19444407 0.46183379 0.18966481]
Cost History: [0.2126848827535125, 0.2096956540023033, 0.2068048415077832, 0.20400913903147738, 0.2013053516475558, 0.19
```

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:

```
# Model Evaluation - RMSE
def rmse(Y, Y_pred):
    """
    This function calculates the Root Mean Squared Error (RMSE).

    Parameters:
    Y : numpy.ndarray
        Array of actual (target) dependent variables (m, ).
    Y_pred : numpy.ndarray
        Array of predicted dependent variables (m, ).

    Returns:
    rmse : float
        The root mean squared error.
    """
    # Step 1: Calculate the squared differences between actual and predicted values
    squared_differences = (Y - Y_pred) ** 2

    # Step 2: Calculate the mean of the squared differences
    mean_squared_error = np.mean(squared_differences)

    # Step 3: Take the square root of the mean squared error
    rmse_value = np.sqrt(mean_squared_error)

    return rmse_value
```

2. R2 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.

```
# Model Evaluation - R-squared
def r2(Y, Y_pred):
    """
    This function calculates the R Squared value, which measures the goodness of fit.

    Parameters:
    Y : numpy.ndarray
        Array of actual (target) dependent variables (m, ).
    Y_pred : numpy.ndarray
        Array of predicted dependent variables (m, ).

    Returns:
    r2 : float
        The R-squared value.
    """
    # Step 1: Calculate the mean of the actual values (Y)
    mean_y = np.mean(Y)

    # Step 2: Calculate the Total Sum of Squares (SST)
    ss_tot = np.sum((Y - mean_y) ** 2)

    # Step 3: Calculate the Sum of Squared Residuals (SSR)
    ss_res = np.sum((Y - Y_pred) ** 2)

    # Step 4: Calculate the R-squared value
    r2_value = 1 - (ss_res / ss_tot)

    return r2_value
```

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 R2 .

• **To - Do:** 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 R2

```
# Gradient Descent Function (as you wrote earlier)
def gradient_descent(X, Y, W, alpha, iterations):
    cost_history = [0] * iterations
    m = len(Y)

    for iteration in range(iterations):
        # Step 1: Hypothesis Values
        Y_pred = np.dot(X, W)  # Predicted Y values
        # Step 2: Difference between Hypothesis and Actual Y
        loss = Y_pred - Y
        # Step 3: Gradient Calculation
        dw = (1/m) * np.dot(X.T, loss)  # Gradient for the weights
        # Step 4: Updating Values of W using Gradient
        W -= alpha * dw
        # Step 5: New Cost Value
        cost = cost_function(X, Y, W)
        cost_history[iteration] = cost

    return W, cost_history

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

# Model Evaluation - RMSE
def rmse(Y, Y_pred):
    return np.sqrt(np.mean((Y - Y_pred) ** 2))

# Model Evaluation - R2
def r2(Y, Y_pred):
    ss_tot = np.sum((Y - np.mean(Y)) ** 2)
    ss_res = np.sum((Y - Y_pred) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    return r2

# Main Function
def main():
    # Step 1: Load the dataset
    data = pd.read_csv('/content/drive/MyDrive/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()
```

```
Final Weights: [0.34811659 0.64614558]
Cost History (First 10 iterations): [2013.165570783755, 1640.286832599692, 1337.0619994901588, 1090.4794892850578, 889.9
RMSE on Test Set: 5.2798239764188635
R-Squared on Test Set: 0.8886354462786421
```

Present your finding:

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

Ans: The model's performance is acceptable, as it shows a high R-squared value of 0.8886, indicating that 88.86% of the variance in the target variable is explained, and a relatively low RMSE of 5.28, suggesting good prediction accuracy. There is no indication of overfitting or underfitting, as the model generalizes well to the test set without performing poorly on either the training or test data.

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

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

# Function to calculate Cost (Mean Squared Error)
def cost_function(X, Y, W):
    m = len(Y)
    Y_pred = np.dot(X, W)
    cost = (1/(2*m)) * np.sum((Y_pred - Y) ** 2)
    return cost

# Gradient Descent function
def gradient_descent(X, Y, W, alpha, iterations):
    cost_history = []
    m = len(Y)
    for _ in range(iterations):
        Y_pred = np.dot(X, W)
        loss = Y_pred - Y
        dw = (1/m) * np.dot(X.T, loss)
        W = W - alpha * dw
        cost = cost_function(X, Y, W)
        cost_history.append(cost)
    return W, cost_history

# Root Mean Squared Error (RMSE)
def rmse(Y, Y_pred):
    return np.sqrt(np.mean((Y - Y_pred) ** 2))

# R-Squared function
def r2(Y, Y_pred):
    ss_tot = np.sum((Y - np.mean(Y)) ** 2)
    ss_res = np.sum((Y - Y_pred) ** 2)
    return 1 - (ss_res / ss_tot)

# Main Function to integrate all steps
def main():
    # Load the dataset
    data = pd.read_csv('/content/drive/MyDrive/student.csv')  # Replace with your actual CSV file
    X = data[['Math', 'Reading']].values  # Features
    Y = data['Writing'].values  # Target

    # Split the dataset 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)

    # Initialize weights, number of iterations, and learning rates
    W_initial = np.zeros(X_train.shape[1])
    iterations = 1000

    # Experiment with different learning rates
    learning_rates = [0.0001, 0.001, 0.01, 0.1, 0.5]

    for alpha in learning_rates:
        print(f"Experimenting with Learning Rate: {alpha}")

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

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

        # Evaluate the model
        model_rmse = rmse(Y_test, Y_pred)
        model_r2 = r2(Y_test, Y_pred)

        # Output the results
        print("Final Weights:", W_optimal)
        print("Final Cost (Last Iteration):", cost_history[-1])
        print("RMSE on Test Set:", model_rmse)
        print("R-Squared on Test Set:", model_r2)
        print("-" * 50)

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

```
Experimenting with Learning Rate: 0.0001
Final Weights: [0.0894932  0.89504864]
Final Cost (Last Iteration): 10.26076310841341
RMSE on Test Set: 4.792607360540954
R-Squared on Test Set: 0.908240340333986
--------------------------------------------------
Experimenting with Learning Rate: 0.001
```

```
Final Weights: [nan nan]
Final Cost (Last Iteration): nan
RMSE on Test Set: nan
R-Squared on Test Set: nan
--------------------------------------------------
Experimenting with Learning Rate: 0.01
Final Weights: [nan nan]
Final Cost (Last Iteration): nan
RMSE on Test Set: nan
R-Squared on Test Set: nan
--------------------------------------------------
Experimenting with Learning Rate: 0.1
Final Weights: [nan nan]
Final Cost (Last Iteration): nan
RMSE on Test Set: nan
R-Squared on Test Set: nan
--------------------------------------------------
Experimenting with Learning Rate: 0.5
Final Weights: [nan nan]
Final Cost (Last Iteration): nan
RMSE on Test Set: nan
R-Squared on Test Set: nan
--------------------------------------------------
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
<ipython-input-21-9b7fc6105cff>:9: RuntimeWarning: overflow encountered in square
  cost = (1/(2*m)) * np.sum((Y_pred - Y) ** 2)
<ipython-input-21-9b7fc6105cff>:20: RuntimeWarning: invalid value encountered in subtract
  W = W - alpha * dw
```

The experiment with different learning rates showed that a very small learning rate (0.0001) resulted in successful training, with reasonable weights and good model performance (low RMSE and high R^2). However, as the learning rate increased (0.001, 0.01, 0.1, 0.5), the model failed, producing NaN values due to numerical instability, which is typical when the learning rate is too high. The large updates during gradient descent caused overflow and invalid calculations. Thus, the optimal learning rate for this model is 0.0001, ensuring stable convergence and effective performance.