Week 5
Workshop
Concept And Technologies Of AI

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

```
df = pd.read_csv('/content/drive/MyDrive/ConceptAndTechnologiesOfAI/Copy of student.csv')
df.head()
```

	Math	Reading	Writing
0	48	68	63
1	62	81	72
2	79	80	78
3	76	83	79
4	59	64	62

Next steps: Generate code with df New interactive sheet

```
df.tail()
```

	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}.

```
df.info()
```

```
<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}

```
df.describe()
```

	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).

```
X = df[["Math", "Reading"]]
Y = df['Writing']
print(X.shape)
print(Y.shape)
```

```
(1000, 2)
(1000,)
```

• 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.

```
import numpy as np

X = df[["Math", "Reading"]].values
Y = df["Writing"].values.reshape(-1, 1)

W = np.zeros((2, 1))

n = X.shape[0]
split_idx = int(0.7 * n)

X_train = X[:split_idx]
Y_train = Y[:split_idx]

X_test = X[split_idx:]
Y_test = Y[split_idx:]

Y_pred = X_train.dot(W)

print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
print(Y_pred[:5])
```

```
(700, 2)
(700, 1)
(300, 2)
(300, 1)
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

To - Do - 4:

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

```
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.
    """
    X = np.array(X, dtype=float)
    Y = np.array(Y, dtype=float).reshape(-1,1)
    W = np.array(W, dtype=float).reshape(-1,1)

    n = len(Y)
    predictions = X.dot(W)
    errors = predictions - Y
    cost = (1 / (2 * n)) * np.sum(errors ** 2)
    return cost
```

```
cost = cost_function(X_train, Y_train, W)
print(cost)
```

```
2474.7264285714286
```

To - Do - 5:

Make sure your code at To - Do - 4 passed the following 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))
```

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

To - Do - 6:

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

```
[92]
✓ 0s
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.
    """
    X=np.array(X, dtype=float)
    Y=np.array(Y, dtype=float).reshape(-1,1)
    W=np.array(W, dtype=float).reshape(-1,1)

    # Initialize cost history
    cost_history = [0] * iterations
    # Number of samples
    m = len(Y)
    W_update=W.copy()
    for iteration in range(iterations):
      # Step 1: Hypothesis Values
      Y_pred = X @ W_update
      # 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_update = W_update-alpha*dw
      # 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:

```
np.random.seed(0)
X = np.random.rand(100, 3)
Y = np.random.rand(100)
W = np.random.rand(3)
final_params, cost_history = gradient_descent(X, Y, W, alpha=0.01, iterations=10)
print("Final Parameters:", final_params)
print("Cost History:", cost_history)

Final Parameters: [[0.38576196]
 [0.91149014]
 [0.08672023]]
Cost History: [np.float64(0.10711197094660153), np.float64(0.10634880599939901), np.float64(0.10559826315680618), np.float64(0.10486012948320558), np.float64(0.1041341956
```

To - Do - 8:

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

```
def rmse(Y, Y_pred):
    """
    Computes Root Mean Squared Error between actual and predicted values.
    """
    Y = np.array(Y, dtype=float).reshape(-1,1)
    Y_pred = np.array(Y_pred, dtype=float).reshape(-1,1)
    return np.sqrt(np.mean((Y_pred - Y) ** 2))
```

To - Do - 9 - Implementation in the Code:
Complete the following code or write your own for r2 loss:

```python
def r2(Y, Y_pred):
    """
    Computes R-squared (coefficient of determination) between actual and predicted values.

    Parameters:
    Y: Actual target values
    Y_pred: Predicted target values

    Returns:
    R-squared value
    """
    Y = np.array(Y, dtype=float).reshape(-1,1)
    Y_pred = np.array(Y_pred, dtype=float).reshape(-1,1)

    ss_res = np.sum((Y - Y_pred) ** 2)
    ss_tot = np.sum((Y - np.mean(Y)) ** 2)

    return 1 - (ss_res / ss_tot)
```

• 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 R2.

Re-write the following code or Write your own:

```python
from sklearn.model_selection import train_test_split
def main():
  # Step 1: Load the dataset
  data = pd.read_csv('/content/drive/MyDrive/ConceptAndTechnologiesOfAI/Copy of 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): [np.float64(2013.165570783755), np.float64(1640.286832599692), np.float64(1337.0619994901588), np.float64(1090.4794892850578), np.floa
RMSE on Test Set: 5.2798239764188635
R-Squared on Test Set: 0.8886354462786421
```

```
To - Do - 11 - Present your finding:

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

Answer: The training error and test error are reasonably close, indicating
that the model generalizes well.
```

```
The RMSE value is low, which means the predicted writing scores are close
to the actual values.

The R² score is positive and reasonably high, showing that Math and
Reading scores explain a good portion of the variance in Writing scores.

The model neither overfits nor underfits the data.
Hence, the model performance is acceptable for a simple linear regression
model without a bias term.
```

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

```python
    # 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: [[nan]
 [nan]]
Cost History (First 10 iterations): [np.float64(23202998.436207462), np.float64(219334644806.26913), np.float64(2073340323269638.8), np.float64(1.959900179065609e+19),
RMSE on Test Set: nan
R-Squared on Test Set: nan
/usr/local/lib/python3.12/dist-packages/numpy/_core/fromnumeric.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/tmp/ipython-input-358009771.py:21: RuntimeWarning: overflow encountered in square
  cost = (1 / (2 * len(Y))) * np.sum(errors ** 2)
/tmp/ipython-input-358009771.py:42: RuntimeWarning: overflow encountered in matmul
  gradient = (1 / m) * (X.T @ loss)
/tmp/ipython-input-358009771.py:43: RuntimeWarning: invalid value encountered in subtract
  W_update = W_update - alpha * gradient
```