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

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.
- Our Recommendation - `Google Colaboratory`.
- 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.`

Figure 1: Starting with Linear Models.

Linear Models are Everywhere!!!

2 Understanding Design Matrix for Linear Regression:

The Design Matrix ($\mathbf{X}$) 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 $\mathbf{X}$ is as follows:

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

- 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:

$$y = Xw + \epsilon$$

Where:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \quad \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}.$$

Parameter Estimation:

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

$$\mathbf{Y} = \mathbf{W}^T \mathbf{X}$$

$$\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}, \quad \text{where } \mathbf{W} \in \mathbb{R}^d$$

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,n} \end{bmatrix}, \quad \text{where } \mathbf{X} \in \mathbb{R}^{d \times n}$$

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \text{where } \mathbf{Y} \in \mathbb{R}^n$$

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

- **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.

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:

$$\mathbf{L}(w) = \frac{1}{2n} \sum_{i=1}^{n} \left(y_{\text{pred}(i)} - y_i \right)^2$$

where:

$$y_{\text{pred}}(w) = W^T \mathbf{X}$$

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:

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

The hypothesis function $h_\theta(X)$ is calculated as:

$$h_\theta(X) = \mathbf{X} \cdot \mathbf{W} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \\ 11 \end{bmatrix}$$

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: $\boxed{0}$

Testing a Cost Function:

```
    # 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))
```

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 $\mathbf{b} = 0$ 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:

$$\frac{\partial \text{Loss}}{\partial w} = 2 \cdot (h_w(x) - y) \cdot x$$

Gradient Descent Update Rule

The Gradient Descent update rule for the weights is:

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

where:

- α is the learning rate,
- 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 $\text{Loss} = (h_w(x) - y)^2$.
   (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:

$$\mathbf{Y}_{\text{pred}} = w_1 \cdot X$$

2. Compute the loss function:

$$\text{loss} = \mathbf{Y}_{\text{pred}} - \mathbf{Y}$$

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.

Implement your code here or write your own:

Gradient Descent from Scratch:

```
 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
```

Testing the function for Gradient Descent:

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)
```

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.

Implementation 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
```

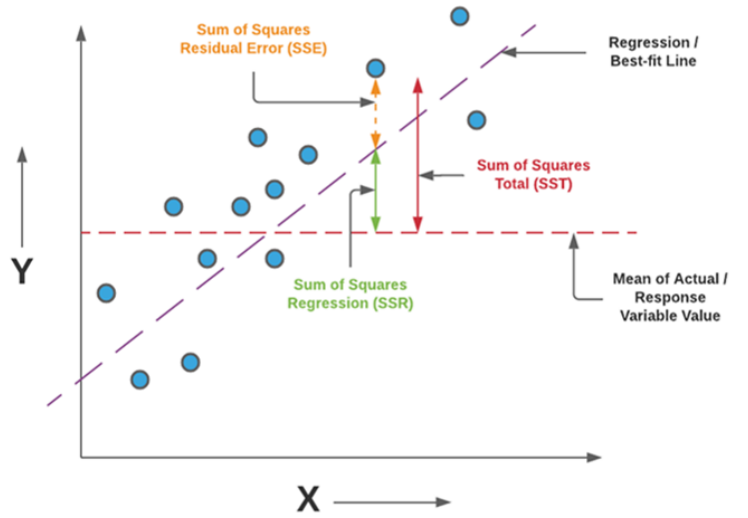

Figure 2: Understanding the Residuals.

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:

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

Where:

$$\text{SSR} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad \text{(Sum of Squared Residuals)}$$

$$\text{SST} = \sum_{i=1}^{n}(y_i - \bar{y})^2 \quad \text{(Total Sum of Squares)}$$

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.

Implementation in the Code:

Complete the following code or write your own:

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
```

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.

- **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 ($\mathbf{X}$) and target vector ($\mathbf{Y}$).
  3. Defines the weight matrix ($\mathbf{W}$) 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()
```

Present your finding:

1. Did your Model Overfitt, Underfitts, or performance is acceptable.
2. Experiment with different value of learning rate, making it higher and lower, observe the result.

The - End