Rachit Manandhar

2501387

np03cs4a240053

[14]:
```python
# Rachit Manandhar
# 2501387
# np03cs4a240053
```

[4]:
```python
import pandas as pd
import numpy as np
```

3.1.1 Data Understanding, Analysis and Preparations:

To - Do - 1:

[5]:
```python
# 1
df = pd.read_csv("student.csv")
```

[21]:
```python
# 2
print("First 5 rows: ")
print(df.head())
print("\nLast 5 rows: ")
print(df.tail())
```

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

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

[22]:
```python
# 3
print("Info: ")
df.info()
```

```
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
```

[23]:
```python
# 4
print("Descriptive info: ")
df.describe()
```

Descriptive info:

[23]:

	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

[6]:
```python
# 5

features_X = df.drop(columns = ["Writing"]).values
label_Y = df["Writing"].values
```

To - Do - 2:

[7]:
```python
X = features_X.T
Y = label_Y

d = X.shape[0]
W = np.zeros((d, 1))

y_pred = (W.T @ X).T

print("X shape:", X.shape)      # (d, n)
print("W shape:", W.shape)         # (d, 1)
print("Y shape:", Y.shape)      # (n, 1)
print("Y_pred shape:", y_pred.shape)
```

```
X shape: (2, 1000)
W shape: (2, 1)
Y shape: (1000,)
Y_pred shape: (1000, 1)
```

To - Do - 3:

[9]:
```python
def train_test_split(X, Y, test_size = 0.2, random_state=42):
    indices = np.arange(len(X))
    np.random.seed(random_state)
    np.random.shuffle(indices)

    split = int(test_size * len(X))

    train_indices = indices[split:]
    test_indices = indices[:split]

    return X[train_indices], X[test_indices], Y[train_indices], Y[test_indices]

x_train, x_test, y_train, y_test = train_test_split(features_X, label_Y, test_size = 0.2, random_state = 42)
```

To - Do - 4:

[10]:

```python
def cost_function(X, Y, W):
    """
    Calculates the Mean Square Error (MSE)

    Arguments:
    X: array-like, shape (n_samples, n_features)
        Feature Maxtrix
    Y: array-like, shape (n_samples,)
        True target values.
    W: array-like, shape (n_features, )
        Weight vector

    Returns:
    float
        The Mean Squared Error (MSE)
    """
    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)

    Y_predicted = X @ W
    error = Y_predicted - Y
    MSE = (1 / (2 * n)) * np.sum(error ** 2)

    return MSE
```

To - Do - 5:

[11]:
```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))
```

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

To - Do - 6:

[12]:
```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.
    """
    X = np.array(X,dtype=float)
    Y= np.array(Y,dtype=float).reshape(-1,1)
    W= np.array(W,dtype=float).reshape(-1,1)

    m= len(Y)
    cost_history = []  # To store cost at each iteration
    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: Update W
        W_update = W_update - alpha * dw

        # Step 5: Compute new cost
        cost = cost_function(X, Y, W_update)
        cost_history.append(cost)

        # # PRINT one line per iteration
        # print(f"Iteration {iteration+1}:")
        # print("  Weights:\n", W_update)
        # print("  Cost:", cost)
        # print("-" * 30)
    return W_update, cost_history
```

To - Do - 7:

[13]:
```python
# 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: [[0.20551667]
 [0.54295081]
 [0.10388027]]
Cost History: [np.float64(0.10711197094660153), np.float64(0.10634880599939901), np.float64(0.10559826315680618), np.float64(0.10486012948320558), np.flo
at64(0.1041341956428534), np.float64(0.10342025583900626), np.float64(0.1027181077540776), np.float64(0.1020275524908062), np.float64(0.1013483945144193
1), np.float64(0.1006804415957737), np.float64(0.1000235047554587), np.float64(0.09937739820884377), np.float64(0.09874193931205609), np.float64(0.098116
94850887098), np.float64(0.09750224927850094), np.float64(0.0968976680842672), np.float64(0.09630303432313951), np.float64(0.09571818027612913), np.float
64(0.09514294105952065), np.float64(0.09457715457692842), np.float64(0.09402066147216397), np.float64(0.09347330508290017), np.float64(0.0929349313951191
3), np.float64(0.09240538899833017), np.float64(0.09188452904154543), np.float64(0.0913722051899995), np.float64(0.09086827358260123), np.float64(0.09037
259279010502), np.float64(0.08988502377398919), np.float64(0.08940542984603007), np.float64(0.08893367662855953), np.float64(0.08846963201539432), np.flo
at64(0.08801316613342668), np.float64(0.08756415130486386), np.float64(0.08712246201010665), np.float64(0.08668797485125508), np.float64(0.08626056851623
207), np.float64(0.08584012374351278), np.float64(0.08542652328745133), np.float64(0.08501965188419301), np.float64(0.0846193962181636), np.float64(0.084
22564488912489), np.float64(0.08383828837978763), np.float64(0.08345721902397185), np.float64(0.08308233097530582), np.float64(0.08271352017645425), np.f
loat64(0.08235068432886682), np.float64(0.08199372286303817), np.float64(0.08164253690927113), np.float64(0.08129702926893387), np.float64(0.080957104386
20353), np.float64(0.08062266832028739), np.float64(0.08029362871811391), np.float64(0.07996989478748553), np.float64(0.0796513772706855), np.float64(0.0
7933798841853089), np.float64(0.07902964196486459), np.float64(0.07872625310147845), np.float64(0.07842773845346054), np.float64(0.07813401605495938), n
p.float64(0.0778450053253578), np.float64(0.0775606270458499), np.float64(0.07728080333641404), np.float64(0.07700545763317514), np.float64(0.07673451466
614989), np.float64(0.07646790043736812), np.float64(0.07620554219936448), np.float64(0.07594736843403344), np.float64(0.07569330883184205), np.float64
(0.07544329427139428), np.float64(0.07519725679934074), np.float64(0.07495512961062821), np.float64(0.07471684702908327), np.float64(0.0744823444883241
2), np.float64(0.0742515585129952), np.float64(0.07402442670031911), np.float64(0.0738008877019607), np.float64(0.07358088120619749), np.float64(0.073364
3479203919), np.float64(0.07315122955375959), np.float64(0.07294146880042966), np.float64(0.07273500932279067), np.float64(0.07253179573511871), np.float
64(0.07233177358748233), np.float64(0.0721348893499193), np.float64(0.07194109039688139), np.float64(0.07175032499194182), np.float64(0.0715625422727614
9), np.float64(0.07137769223630935), np.float64(0.07119572572433286), np.float64(0.07101659440907385), np.float64(0.07084025077922623), np.float64(0.0706
66648126131), np.float64(0.07049574053020462), np.float64(0.07032748284759716), np.float64(0.07016183069707572), np.float64(0.0699987404471299), np.float
64(0.06983816920329523), np.float64(0.06968007479569092), np.float64(0.06952441576676843), np.float64(0.06937115135926715), np.float64(0.0692202415043737
5), np.float64(0.06907164681008185), np.float64(0.06892532854974835), np.float64(0.0687812486508435), np.float64(0.06863936968389095), np.float64(0.06849
965485159508), np.float64(0.06836206797815195), np.float64(0.06822657349874123), np.float64(0.06809313644919561), np.float64(0.067961722455845), np.float
64(0.06783229772553254), np.float64(0.06770482903579932), np.float64(0.06757928372523506), np.float64(0.06745562968399212), np.float64(0.0673338353444596
9), np.float64(0.06721386967209597), np.float64(0.06709570215641501), np.float64(0.06697930280212627), np.float64(0.06686464212042395), np.float64(0.0667
5169112042348), np.float64(0.0666404213007429), np.float64(0.06653080464122665), np.float64(0.06642281359480932), np.float64(0.06631642107951677), np.flo
at64(0.06621160047060279), np.float64(0.06610832559281864), np.float64(0.06600657071281309), np.float64(0.0659063105316614), np.float64(0.065807520177520
23), np.float64(0.06571017519840698), np.float64(0.06561425155510119), np.float64(0.06551972561416586), np.float64(0.06542657414108709), np.float64(0.065
33477429352925), np.float64(0.06524430361470467), np.float64(0.06515514002685512), np.float64(0.06506726182484374), np.float64(0.06498064766985515), np.f
loat64(0.06489527658320228), np.float64(0.06481112794023773), np.float64(0.06472818146436811), np.float64(0.0646464172211699), np.float64(0.0645658156126
0431), np.float64(0.06448635737133043), np.float64(0.0644080235551142), np.float64(0.06433079554133217), np.float64(0.06425465502156798), np.float64(0.06
417958399630046), np.float64(0.06410556476968135), np.float64(0.06403257994440141), np.float64(0.0639606124166433), np.float64(0.06388964537111992), np.f
loat64(0.06381966227619645), np.float64(0.06375064687909507), np.float64(0.06368258320118077), np.float64(0.06361545553332655), np.float64(0.063549248431
35755), np.float64(0.06348394671157162), np.float64(0.06341953544633615), np.float64(0.06335599995975896), np.float64(0.06329332582343267), np.float64(0.
06323149885225086), np.float64(0.06317050510029515), np.float64(0.06311033085679153), np.float64(0.06305096264213547), np.float64(0.06299238720398384), n
p.float64(0.0629345915134133), np.float64(0.06287756276114324), np.float64(0.06282128835382297), np.float64(0.0627657559103815), np.float64(0.06271095325
843898), np.float64(0.06265686843077901), np.float64(0.06260348966188053), np.float64(0.06255080538450809), np.float64(0.06249880422636036), np.float64
(0.06244747500677472), np.float64(0.06239680673348793), np.float64(0.06234678859945137), np.float64(0.06229740997970036), np.float64(0.0622486604282762),
np.float64(0.06220052967520031), np.float64(0.062153007623499706), np.float64(0.062106084346282515), np.float64(0.062059750083863094), np.float64(0.06201
399524093575), np.float64(0.06196881038379625), np.float64(0.061924186237610215), np.float64(0.06188011368372787), np.float64(0.0618365837570441), np.flo
```

To - Do - 8:

[14]:
```python
def rmse(y, y_pred):
    """
    Calculates the Root Mean Squred Error (RMSE) between actual and predicted values.

    Arguments:
    y: array-like
        Array of actual (target) values.
    y_pred: array-like
        Array of predicted values.

    Returns:
    float
        The root mean squared error.
    """
    Y = np.array(y, dtype = float).flatten()
    Y_pred = np.array(y_pred, dtype = float).flatten()

    rmse = np.sqrt(np.mean((Y - Y_pred) ** 2))
    return rmse
```

To - Do - 9:

[15]:
```python
def r2(Y, Y_pred):
    """
    This function calculates the R Squared Error.

    Arguments:
    Y: array-like
        Array of actual (target) dependent values.
    Y_pred: array-like
        Array of predicted dependent values.

    Returns:
    float
        R Squared error.
    """
    Y = np.array(Y, dtype=float).flatten()
    Y_pred = np.array(Y_pred, dtype=float).flatten()

    mean_y = np.mean(Y)  # Mean of actual values

    # Total sum of squares
    ss_tot = np.sum((Y - mean_y) ** 2)

    # Sum of squared residuals
    ss_res = np.sum((Y - Y_pred) ** 2)

    # R squared
    r2_score = 1 - (ss_res / ss_tot)

    return r2_score
```

To - Do - 10:

[18]:

```python
# 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)
```

```
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.float64(889.9583270083234), np.float64(726.8940993009545), np.float64(594.2897260808594), np.float64(486.4552052951635), np.float64(398.7634463599484), np.float64(327.4517147324688)]
RMSE on Test Set: 5.2798239764188635
R-Squared on Test Set: 0.8886354462786421
```

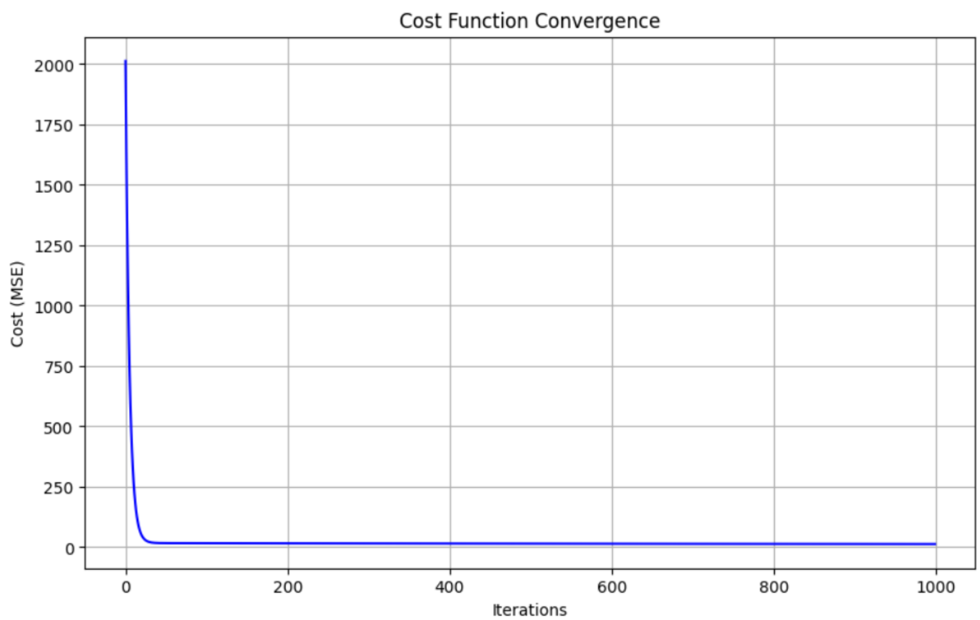
To - Do - 11:

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

```
import matplotlib.pyplot as plt

def plot_cost(cost_history):
    plt.figure(figsize=(10, 6))
    plt.plot(range(len(cost_history)), cost_history, color='blue')
    plt.title('Cost Function Convergence')
    plt.xlabel('Iterations')
    plt.ylabel('Cost (MSE)')
    plt.grid(True)
    plt.show()

plot_cost(cost_history)
```



The model's performance is acceptable and the cost function decreases smoothly indicating proper convergence. The RMSE value being around 5.28 shows that the prediction error is low and R-squared value of approx 0.89 indicates that the model explains most of the variance in writing marks. Therefore, the model neither underfits nor overfits the data.

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

```
learning_rates = [0.000001, 0.00001, 0.0001]

for alpha in learning_rates:
    print("\nLearning rate:", alpha)

    W = np.zeros(X_train.shape[1])
    iterations = 1000

    W_optimal, cost_history = gradient_descent(
        X_train, Y_train, W, alpha, iterations
    )

    Y_pred = X_test @ W_optimal

    print("  Final Cost:", cost_history[-1])
    print("  RMSE:", rmse(Y_test, Y_pred))
    print("  R²:", r2(Y_test, Y_pred))
```

```
Learning rate: 1e-06
  Final Cost: 16.535602355147176
  RMSE: 5.856694748793876
  R²: 0.8629707528684534

Learning rate: 1e-05
  Final Cost: 13.150619992105618
  RMSE: 5.2798239764188635
  R²: 0.8886354462786421

Learning rate: 0.0001
  Final Cost: 10.26076310841341
  RMSE: 4.792607360540954
  R²: 0.908240340333986
```

According to the data above, as the learning rate increases from 0.000001 to 0.0001, the model shows improved convergence with decrease in final cost and RMSE and increase in R-Squared. Therefore, smaller learning rate results in slower convergence while moderate learning rate results in quicker covergence and is more effective but with bigger learning rate it might start to diverge.