Assignment-2: Logistic Regression

Rashik Iram Chowdhury (2111336642), Zarin Akter (2011704042) and Md. Mutasim Farhan (2013123642)
Emails: {rashik.chowdhury@northsouth.edu}, {zarin.akter@northsouth.edu}, {mutasim.farhan@northsouth.edu}

December 19, 2024

I. Introduction

In this assignment, we're delving deeper into Multivariate Linear Regression and Logistic Regression, integrating regularization techniques using gradient descent. Starting with Multivariate Linear Regression on the Air Quality Dataset [1], we employed regularization to counter overfitting by reducing bias and thus enhancing model generalization. By penalizing complex models, regularization methods help prevent fitting noise in training data, resulting in more straightforward models. We evaluate model performance using Mean Squared Error (MSE) on both training and testing data. Furthermore, we apply Logistic Regression on the Smarket Dataset, addressing a binary classification task. By evaluating the Mean Square Error (MSE) and accuracy of training and testing datasets, we aim to demonstrate regularization's role in strengthening model reliability and adaptability across diverse datasets. The paper is structured into three sections. Section [2] outlines the methodology employed for both linear and logistic regression analyses. In Section [3], the experimental findings are presented and analyzed. Finally, Section [4] offers conclusions drawn from the study's results and discusses their implications.

II. Method

The methodology for both problems shares the same approach. However, there are slight differences in the hypothesis functions. As a result, the regularized cost functions differ slightly from each other, as depicted in Equation (1-2).

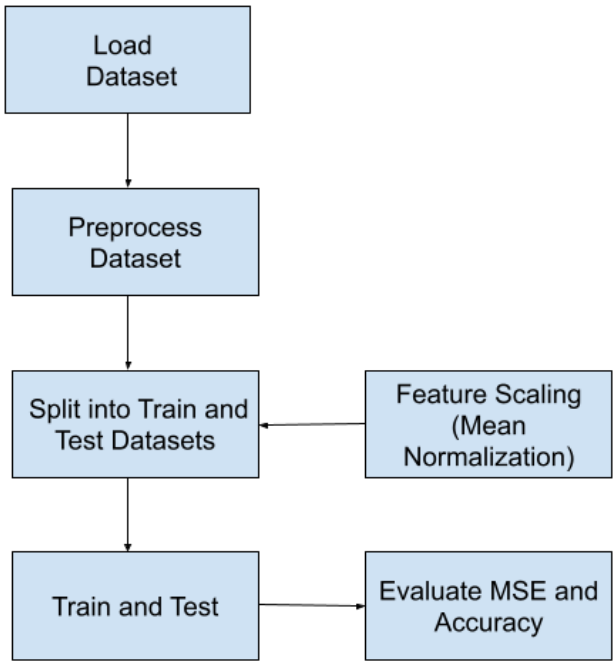

Fig. 1. Methodology Block Diagram

A. Dataset and Attributes

1) Data pre-processing:

- **Load Dataset:** At first, we downloaded both the datasets AirQuality and Smarket in CSV format in separate Jupyter Notebooks and loaded these formatted datasets using pandas library. Then, the datasets were converted to a numpy array.

```
import pandas as pd
import numpy as np
df = pd.read_csv("G:\\Assignment\\AirQualityUCI.csv")
data = np.array(df)
```

```
import pandas as pd
import numpy as np
df = pd.read_csv("G:\\Assignment\\Smarket.csv")
data = np.array(df)
```

- **Preprocess Dataset:** We only found missing values in the AirQuality dataset, which were labeled as -200. We have replaced -200 with the mean of the entire respective columns in the AirQuality dataset. Then, we dropped the first two columns as they were not applicable for linear Regression.
  For logistic regression, we have converted the categorical target- Up and Down into 1 and 0, respectively. Otherwise, binary classification would not be possible, as logistic regression expects categorical data to be converted into numerical data. We did not remove any feature as a feature selection because these features carried useful information, and thus, to avoid under-fitting (high bias) for this dataset, the entire dataset was considered for better representation.

B. Linear Regression Code

```
import math
final_dataset = data[:,2:15]
#Replacing Missing Values by mean
total_sum = [0] * len(final_dataset[0])
instances = [0] * len(final_dataset[0])
for i in range(len(total_sum)):
    total_sum[i] = 0
    instances[i] = 0
for columns in range(len(final_dataset[0])):
    for rows in range(len(final_dataset)):
        x = final_dataset[rows][columns]
        if x != -200:
            total_sum[columns] += x
            instances[columns] += 1
means = [round(total_sum / instances,1) if instances != 0 else 0 for
total_sum, instances in zip(total_sum, instances)]
for columns in range(len(final_dataset[0])):
    for rows in range(len(final_dataset)):
        if final_dataset[rows][columns] == -200:
            final_dataset[rows][columns] = means[columns]
```

C. Logistic Regression Code

```
import math
final_dataset = data[:,:]
target = 8  # Assuming the target column index is the 8th column
for row in range(len(final_dataset)):
    x = final_dataset[row][target]
    if x == "Up":
        final_dataset[row][target] = 1
    elif x == "Down":
        final_dataset[row][target] = 0
```

- **Split into Train and Test Datasets:** We then split both datasets into train and test datasets at a ratio of 75:25, respectively. The Smarket dataset had 8 features (X) and 1 target (y). In the AirQuality dataset, there were 13 features. Among the remaining 13 features, the first 10 were considered features (X), and the last column was considered a target (y). The remaining dataset columns will later be considered targets for evaluating the model's performance, as shown in the Experiment Results in section [III].

D. *Linear Regression Code*

```
import math
final_dataset = data[:,2:15]
total_instances = len(final_dataset)
#We could have tried shuffling for a better experiment
Train_size = math.floor(0.75*total_instances)#7017
Train_ds = final_dataset[:Train_size]
Test_ds = final_dataset[Train_size:]
```

E. *Logistic Regression Code*

```
import math
final_dataset[:,8] #our target column for now
total_instances = len(final_dataset)
#We could have tried shuffling for a better experiment
Train_size = math.floor(0.75*total_instances)#937
Train_ds = final_dataset[:Train_size]
Test_ds = final_dataset[Train_size:]
```

F. *Feature Scaling (Mean Normalization)*

After replacing the dataset with the mean value, we calculated each column's standard deviation to perform mean normalization and replaced the original values with their respective mean normalized values. It is important to note that for the Smarket dataset, we are not considering the target 'Direction' for normalization as we want to work with binary classification, and the values have already been changed to 0 and 1. This feature scaling technique is applied to give equal importance to the data for each feature. The feature scaling function returned the mean and standard deviation from the training dataset. This standard deviation and mean are used to directly calculate the normalized values for the test dataset and override its existing values.

G. *Linear Regression Code*

```
def feature_scaling(dataset):
    total_sum = [0] * len(dataset[0])
    instances = [0] * len(dataset[0])

    for columns in range(len(dataset[0])):
        for rows in range(len(dataset)):
            x = dataset[rows][columns]
            if x != -200:
                total_sum[columns] += x
                instances[columns] += 1

    means = [total_sum[i] / instances[i] if instances[i] != 0 else 0 for i
    in range(len(total_sum))]

    sum_squared_diff = [0] * len(dataset[0])

    for columns in range(len(dataset[0])):
        for rows in range(len(dataset)):
            x = dataset[rows][columns]
            diff = x - means[columns]
            sum_squared_diff[columns] += diff ** 2

    std_deviations = [((sum_squared_diff[i] / (instances[i] - 1)) ** 0.5)
    if instances[i] > 1 else 0 for i in range(len(sum_squared_diff))]

```

```
    for columns in range(len(dataset[0])):
        for rows in range(len(dataset)):
            value = dataset[rows][columns]
            normalized_value = (value - means[columns]) / std_deviations[columns]
            if std_deviations[columns] != 0 else 0
            dataset[rows][columns] = normalized_value

    return dataset, means, std_deviations
```

H. Logistic Regression Code

```
def feature_scaling(dataset):
    total_sum = [0] * len(dataset[0])
    instances = [0] * len(dataset[0])

    for columns in range(len(dataset[0])-1):
        for rows in range(len(dataset)):
            x = dataset[rows][columns]
            total_sum[columns] += x
            instances[columns] += 1

    means = [total_sum[i] / instances[i] if instances[i] != 0 else 0 for i
    in range(len(total_sum))]

    sum_squared_diff = [0] * len(dataset[0])

    for columns in range(len(dataset[0])-1):
        for rows in range(len(dataset)):
            x = dataset[rows][columns]
            diff = x - means[columns]
            sum_squared_diff[columns] += diff ** 2

    std_deviations = [((sum_squared_diff[i] / (instances[i] - 1)) ** 0.5)
    if instances[i] > 1 else 0 for i in range(len(sum_squared_diff))]

    for columns in range(len(dataset[0])-1):
        for rows in range(len(dataset)):
            value = dataset[rows][columns]
            normalized_value = (value - means[columns]) / std_deviations[columns]
            if std_deviations[columns] != 0 else 0
            dataset[rows][columns] = normalized_value

    return dataset, means, std_deviations
```

- **Train and Test:** We calculated the mean squared error or cost function after feature scaling and splitting test and train datasets into X_train, y_train, X_test, and y_test.
  1) Multivariate Linear Regression: As the training dataset has ten features, weight is considered a vector, and bias is a scalar number. Each instance or row is multiplied with the respective weights as a dot product and added with a bias to compute the hypothesis function in Equation 3. This hypothesis function engenders a predicted target value, which is deducted from the actual value. This difference is then squared over the number of instances in the dataset to calculate the mean squared difference of the hypothesis. Furthermore, a regularizer is added, traversing through all the weights and engenders a smaller value by multiplying the square of each weight by the regularization parameter λ. Then, both the total cost and total regularized value are added and divided by twice the number of instances. Similarly, partial derivatives with respect to the weights and bias term were calculated, where for weights, only the partial derivative of the regularizer was added to compute the gradient descent algorithm, which takes in 8 parameters. We initialized the weight and bias values using the correlation heatmap in Fig. 3. The gradient descent algorithm returns the optimal weights and bias vector terms values for which the cost function or mean squared error is minimal over the number of epochs or iterations.

I. Linear Regression Code

```
def cost_function_with_regularization(X, y, w, b, lamb_da):

    m = X.shape[0]  #This is the number of instances in the dataset
    n = X.shape[1]
    cost = 0
    regularizer = 0
    for i in range(m):
        f_wb = np.dot(X[i], w) + b # This is the hypothesis function
        cost = cost + (f_wb - y[i])**2

    for j in range(n):
        regularizer = regularizer + w[j] ** 2

    total_regularization = regularizer * lamb_da / (2 * m)
    total_cost = 1 / (2 * m) * cost

    regularized_cost = total_cost + total_regularization
    return regularized_cost

import copy
def gradient_calculation(X, y, w, b, lamb_da):

    m,n = X.shape  #Here m is number of instances and n is number of features
    dj_dw = np.zeros((n,))
    dj_db = 0.0

    for i in range(m):
        difference = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + difference * X[i, j]
        dj_db = dj_db + difference

    dj_dw = (dj_dw + lamb_da*w) / m
    dj_db = dj_db / m

    return dj_db, dj_dw

 def gradient_descent(X, y, w_in, b_in, cost_function, gradient_calculation,
 alpha, iterations):

    J_history = []
    w = copy.deepcopy(w_in)
    b = b_in

    for i in range(iterations):

        dj_db,dj_dw = gradient_calculation(X, y, w, b, lamb_da)
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Save cost J at each iteration
        if i<50000:      # prevent resource exhaustion
            J_history.append(cost_function(X, y, w, b, lamb_da))

        if i% math.ceil(iterations / 10) == 0:
            print(f"Iteration␣{i:4d}:␣Cost␣{J_history[-1]:8.2f}␣␣␣")
```

```
57
58      return w, b, J_history #return final w,b and J history for graphing
```

1) Logistic Regression: In the case of logistic regression, only the hypothesis is different as it uses a sigmoid function to generate outputs from 0 to 1. A threshold value of 0.5 is considered to classify the predicted output as 1 or 0. Similarly, as the multivariate training dataset has 8 features, thus weight is considered as a vector and bias as a scalar number. This combined input goes to the sigmoid function, as shown in Equation (3-4). Another difference between both classifiers is that they have different Cost functions, as shown in Equation (1-2). The logarithm is considered to create a convex-shaped output to reach global minima. Otherwise, it would reach local minima. The resulting partial derivatives and gradient descent are the same for both, as shown in Equation (5-7).

J. Logistic Regression Code

```
1  def cost_function_with_regularization(X, y, w, b, lamb_da):
2      m = X.shape[0]  # Number of instances in the dataset
3      n = X.shape[1]  # Number of features
4      cost = 0
5      regularizer = 0
6
7      for i in range(m):
8          z = np.dot(X[i], w) + b
9          f_wb = 1 / (1 + np.exp(-z))  # Hypothesis function
10         cost += (y[i] * np.log(f_wb) + (1 - y[i]) * np.log(1 - f_wb))
11
12     for j in range(n):
13         regularizer += w[j] ** 2
14
15     total_regularization = regularizer * lamb_da / (2 * m)
16     total_cost = (-1 / m) * cost
17     regularized_cost = total_cost + total_regularization
18     return regularized_cost
19
20     def gradient_calculation(X, y, w, b, lamb_da):
21
22     m,n = X.shape  #Here m is number of instances and n is number of features
23     dj_dw = np.zeros((n,))
24     dj_db = 0.0
25
26     for i in range(m):
27         z = np.dot(X[i], w) + b
28         f_wb = 1 / (1 + np.exp(-z))
29         difference = f_wb - y[i]
30         for j in range(n):
31             dj_dw[j] = dj_dw[j] + difference * X[i, j]
32         dj_db = dj_db + difference
33
34     dj_dw = (dj_dw + lamb_da*w) / m
35     dj_db = dj_db / m
36     return dj_db, dj_dw
37
38    def gradient_descent(X, y, w_in, b_in, lamb_da_in, cost_function, gradient_calculation, alp
39
40     J_history = []
41     w = copy.deepcopy(w_in)
42     b = b_in
43     lamb_da = lamb_da_in
44     for i in range(iterations):
45
```

```
        dj_db,dj_dw = gradient_calculation(X, y, w, b, lamb_da)

        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Save cost J at each iteration
        if i<50000:      # prevent resource exhaustion
            J_history.append(cost_function(X, y, w, b, lamb_da))

        if i% math.ceil(iterations / 10) == 0:
            print(f"Iteration␣{i:4d}:␣Cost␣{J_history[-1]:8.2f}␣␣␣")

    return w, b, J_history #return final w,b and J history for graphing
```

- **Evaluate MSE:** After utilizing the gradient descent algorithm to find optimal weights and bias values, the regularized cost function was used, where these updated values were given as inputs. The corresponding MSE outputs for the train and test datasets were calculated and printed for both problems. For the AirQuality dataset, to better analyze our hypothesis or model, we used the same weights and bias values as inputs to the cost function for the other two target features separately after updating the y_train and y_test to compute MSE for both targets. For the Smarket dataset, accuracy was also calculated to better comprehend the model's performance after calculating the MSE values for train and test.

K. Regression Classifier

1) Linear Regression Cost Function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1}$$

2) Logistic Regression Cost Function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{2}$$

Here, $J(\theta)$ represents the cost function, m is the number of training examples, $h_\theta(x)$ is the hypothesis function, $y^{(i)}$ represents the actual output for the i-th training example, $x^{(i)}$ represents the input features for the i-th training example, θ_j represents the parameters (weights) of the model, n is the number of features, and λ is the regularization parameter.

3) Logistic Regression Hypothesis Function:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}} \tag{3}$$

where,

$$g(\theta^T x) = g(z) = \frac{1}{1+e^{-z}} \tag{4}$$

Here $h_\theta(x)$ refers to the logistic regression hypothesis, x is the input features, and $g(\theta^T x)$ is the sigmoid function.

4) Gradient descent algorithm:

$$\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}\right] + \frac{\lambda}{m}\theta_1; \qquad \text{[when } j=1\text{]}$$

$$\vdots$$

$$\theta_n := \theta_n - \alpha\frac{1}{m}\sum_{i=1}^{m}\left[(h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)}\right] + \frac{\lambda}{m}\theta_n; \qquad \text{[when } j=n\text{]}$$

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}); \qquad \text{[when } j=0\text{(bias)]}$$

This equation represents the update rule for each parameter θ_j in the gradient descent algorithm, where α is the learning rate, m is the number of training examples, $h_\theta(x^{(i)})$ is the hypothesis function, $y^{(i)}$ is the actual output for the i-th training example, and $x_j^{(i)}$ is the j-th feature of the i-th training example.

III. Experiment results

A. Data Analysis

1) Data correlation analysis: A data correlation analysis examines the strength and direction of a relationship between two or more variables in a dataset. It assesses how changes in one variable relate to changes in another.

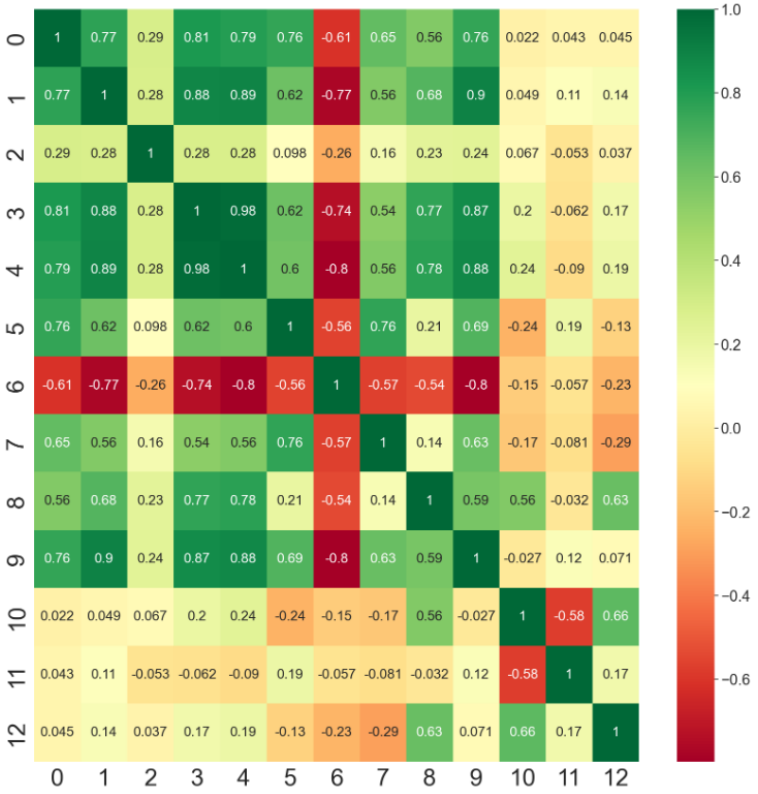

Fig. 2. Correlation between attributes for multivariate linear regression

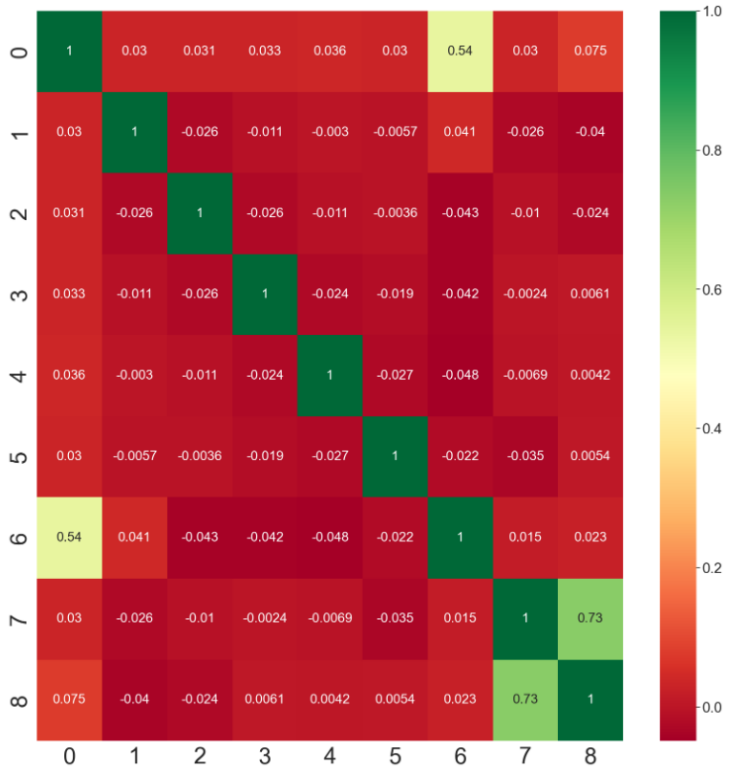

Fig. 3. Correlation between attributes for logistic regression

A correlation heatmap is a visual tool that shows the correlation between many variables as a matrix with different colors. Similar to a color chart, it illustrates how closely various variables are related. The heatmap in Fig. 3 displays the degree of correlation between the features in the dataset utilized in this study. It shows how the independent variables are correlated.

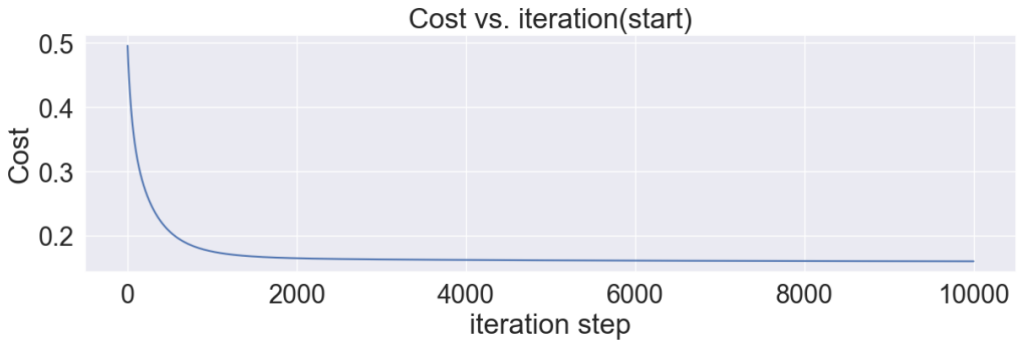

Fig. 4. Linear Regression Cost vs Iteration

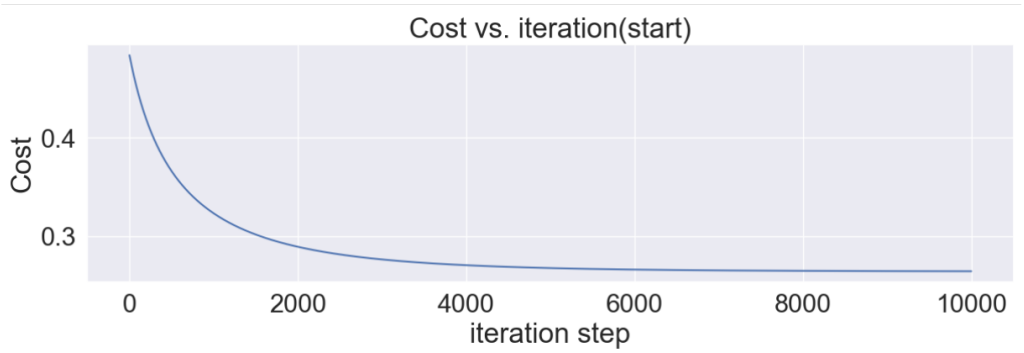

Fig. 5. Logistic Regression Cost vs Iteration

The cost over iteration graph shows the training performance of the model by displaying the Mean Squared Error or cost after each iteration.

TABLE I
VARIATION OF MEAN SQUARED ERROR (MSE/COST) WITH THE CHANGE OF LEARNING RATE AND ITERATION FOR LINEAR REGRESSION

Trials	**Learning rate (α)**	**Iterations**	**MSE (Cost)**
1	0.03	1000	0.16
2	0.03	5000	0.16
3	0.001	3000	0.25
4	0.1	1000	0.16
5	0.01	1000	0.18
6	0.01	10000	0.16

TABLE II
MEAN SQUARED ERROR FOR TARGETS AH, RH, AND T FOR LINEAR REGRESSION

MSE					
AH		RH		T	
Train	Test	Train	Test	Train	Test
0.160	0.086	0.674	1.127	0.493	0.312

TABLE III
VARIATION OF MEAN SQUARED ERROR (MSE/COST) WITH THE CHANGE OF LEARNING RATE AND ITERATION FOR LOGISTIC REGRESSION

Trials	**Learning rate (α)**	**Iterations**	**MSE (Cost)**
1	0.03	1000	0.28
2	0.03	5000	0.26
3	0.001	3000	0.40
4	1	1000	0.26
5	0.1	1000	0.26
6	0.01	10000	0.26

TABLE IV
VARIATION OF MEAN SQUARED ERROR (MSE/COST) WITH THE CHANGE OF REGULARIZATION PARAMETER FOR LOGISTIC REGRESSION

Trials	λ	**MSE**	
		Train	**Test**
1	1	0.200	0.312
2	10	0.264	0.504
3	100	0.906	2.427
4	1000	7.328	21.651

Accuracy is a widely used measure to evaluate how well a classification model performs. It simply calculates the proportion of correctly predicted instances out of the total number of cases in the dataset.

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total instances predicted}} \tag{5}$$

TABLE V
MEAN SQUARED ERROR AND ACCURACY FOR TRAIN AND TEST FOR LOGISTIC REGRESSION

MSE		**Accuracy**	
Train	**Test**	**Train**	**Train**
0.264	0.505	95.09%	82.43%

IV. DISCUSSION

In this assignment, we conducted a comprehensive analysis encompassing the Mean Squared Error (MSE) calculation for both the Air Quality and Smarket Datasets. Additionally, we assessed the accuracy specifically for the Smarket Dataset. Employing a gradient descent algorithm, we iteratively optimized weights and bias values. Moreover, we incorporated regularization techniques into the cost function and its gradient derivative to mitigate overfitting.

After rigorous experimentation with various regularization parameters, we identified an optimal setting that yielded superior Mean Squared Error performance. For the linear regression task with the AirQuality dataset, our gradient descent algorithm, configured with a learning rate of 0.01, 10,000 iterations, and a regularization parameter 10, produced exact hypothesis functions. Notably, the resulting MSE values for both training (0.160) and testing (0.079) datasets approached zero, indicating the model's remarkable accuracy. The MSE scores for all three target variables (AH, RH, T) on unseen data were also insignificantly close to zero, affirming the model's robustness.

Similarly, in the logistic regression scenario using the Smarket dataset, our gradient descent algorithm, under the same configuration parameters but with a regularization parameter of 1, generated hypothesis functions of commendable accuracy. The MSE values for the training (0.200) and testing (0.312) datasets were markedly minimized. Additionally, achieving an accuracy of 95.09% on the training dataset and 82.43% on the test dataset demonstrates the model's substantial predictive capability.

Overall, through meticulous experimentation and parameter tuning, we've established highly accurate models for both linear and logistic regression tasks, underscoring the effectiveness of our approach in data analysis and predictive modeling.

REFERENCES

[1] Air quality data set. 2004. Accessed in 2004. https://archive.ics.uci.edu/dataset/360/air+quality.