Assignment-2: Regularization

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:
rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

I. Introduction

This assignment aims to perform regularization on a multi-variable linear regression model for predicting relative humidity (RH) based on the air quality factors in an Italian city. And Perform regularization on Multivariate logistic regression using gradient descent. We used the **Smarket.csv** dataset for the binary classification problem and reported accuracy for the training and testing dataset.

Regularization: Regularization is a technique used in linear regression (and other machine learning models) to prevent overfitting and improve the model's generalisation ability. Overfitting occurs when the model learns to fit the training data too closely, capturing noise and irrelevant patterns that do not generalize well to unseen data.

Regularization introduces a penalty term to the loss function, discouraging overly complex models by penalizing large coefficient values.

$$\text{Before: } f(x) = 28x - 385x^2 + 39x^3 - 174x^4 + 100$$

$$\text{After: } f(x) = 13x - 0.23x^2 + 0.000014x^3 - 0.0001x^4 + 10$$

Cost functoins

The regularized cost function for linear regression is given by:

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

The regularized cost function for logistic regression is given by:

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

Gradient Descent Update Rule is given by:

$$\theta_j := \theta_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right)$$

II. Method

A. Data Exploration and Preprocessing

- For logistic regression, we have preprocessed the **Smarket.csv** file a little bit. We have dropped the column year as we believe it will simplify our work (otherwise we have to use one-hot encoding to separate).
  Our target column values are ”up” and ”down”. We have to change it to a numeric value. So we replace up as 1 and down as 0.

```
import pandas as pd
import numpy as np
data = pd.read_csv('/content/gdrive/MyDrive/data/Smarket.csv')
df = data.drop(['Year'], axis=1)
df.head()
for column in df.columns:
    total = df[column].isnull().sum()
    print(f"Column ‘{column}’ has {total} null values")
    print("\n")
df['Direction']=df['Direction'].replace({'Up': 1, 'Down': 0})
print(df)
df['Direction'] = df['Direction'].astype('float64')
df.info()
df['Direction'] = df['Direction'].astype('int64')
```

B. *Cost Function(Linear Regression) :compute_cost_linear_reg(X, y, w, b, lambda_ = 1)*

- We have worked on assignment 1’s linear regression code. There are some changes in the code. Before, it was only the M.S.E term; this time, it has been added to the regularized term. We did not regularise the term b.

Regularization Term

It calculates the regularization term to penalize large weights, which helps prevent overfitting. The regularization term is the sum of squared weights w_j^2. It’s multiplied by $\frac{\lambda}{2m}$, where λ is the regularization parameter.

```
def compute_cost_linear_reg(X, y, w, b, lambda_ = 1):
    m  = X.shape[0]
    n  = len(w)
    cost = 0.0
    for i in range(m):
        f_wb = np.dot(X[i], w) + b
        cost = cost + (f_wb - y[i])**2
    cost = cost / (2 * m)
    regg_term = 0.0
    for j in range(n):
        regg_term = regg_term + (w[j]**2)
    regg_term = (lambda_/(2*m)) * regg_term
    total_cost = cost + regg_term
    return total_cost
```

C. *Cost Function(Logistic Regression) :compute_cost_logistic_reg(X, y, w, b, lambda_ = 1)*

- We have worked on the given logistic regression code. For regularization, There are some changes in the code. We have to add the regularized term with the given code. And we did not regularise the term b.

REGULARIZATION TERM

It calculates the regularization term to penalize large weights, which helps prevent overfitting. The regularization term is the sum of squared weights w_j^2. It's multiplied by $\frac{\lambda}{2m}$, where λ is the regularization parameter.

```
def sigmoid(z):
    g = 1/(1+np.exp(-z))
    return g
def compute_cost_logistic_reg(X, y, w, b, lambda_= 1):
    m, n = X.shape
    cost = 0.0
    for i in range(m):
        z = np.dot(X[i],w) + b
        f_wb = sigmoid(z)
        cost += -y[i]*np.log(f_wb) - (1-y[i])*np.log(1-f_wb)
    cost = cost/m
    regg_term = 0.0
    for j in range(n):
        regg_term = regg_term + (w[j]**2)
    regg_term = (lambda_/(2*m)) * regg_term
    total_cost = cost + regg_term
    return total_cost
```

D. *Gradient Descent(Linear Regression) : compute_gradient_linear_reg(X, y, w, b,lambda_)*

- As with the cost function, the code has some changes.

REGULARIZATION TERM

It adds the regularization term to the gradients of the weights (dj_dw) to penalize large weights. For each feature j, it adds $\frac{\lambda}{m} \cdot w[j]$ to dj_dw[j].

```
def compute_gradient_linear_reg(X, y, w, b, lambda_):
    m = X.shape[0] # number of examples
    n = X.shape[1] # number of features
    dj_dw = np.zeros((n,))
    dj_db = 0.0
    for i in range(m):
        f_wb = np.dot(X[i], w) + b
        common = f_wb - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + common * X[i, j]
        dj_db = dj_db + common
    dj_dw = dj_dw / m
    dj_db = dj_db / m
    for j in range(n):
        dj_dw[j] = dj_dw[j] + (lambda_/m) * w[j]
    return dj_db, dj_dw
```

E. Gradient Descent(Logistic Regression) : compute_gradient_logistic_reg(X, y, w, b,lambda_)

- As with the cost function, the code has some changes.

REGULARIZATION TERM

It adds the regularization term to the gradients of the weights (dj_dw) to penalize large weights. For each feature j, it adds $\frac{\lambda}{m} \cdot w[j]$ to dj_dw$[j]$.

```
def compute_gradient_logistic_reg(X, y, w, b, lambda_=None):
    m, n = X.shape
    dj_dw = np.zeros(w.shape)
    dj_db = 0.0
    ### START CODE HERE ###
    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)
        err_i  = f_wb_i  - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m
    dj_db = dj_db/m
    for j in range(n):
        dj_dw[j] = dj_dw[j] + (lambda_/m) * w[j]
    return dj_db, dj_dw
```

F. Gradient Function : gradient_descent()

1) Loop over a specified number of iterations. with the given lambda (λ)
2) Compute the gradient of the cost function concerning the parameters using the provided `gradient_function`.
3) Update the parameters (w and b) using the gradient and the learning rate(α).
4) Return the final optimized parameters (w and b), the history of cost values, and the history of parameter values.

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function,
                alpha, num_iters, lambda_):
    m = len(X)
    J_history = []
    w_history = []
    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X, y, w_in, b_in, lambda_)

        w_in = w_in - alpha * dj_dw
        b_in = b_in - alpha * dj_db

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            cost =  cost_function(X, y, w_in, b_in, lambda_)
            J_history.append(cost)
        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0 or i == (num_iters-1):
            w_history.append(w_in)
            print(f"Iteration␣{i:4}:␣Cost␣{float(J_history[-1]):8.2f}␣␣␣")
    return w_in, b_in, J_history, w_history
```

G. *Train Test Split*

- As per the instruction, the first 75% of the data was used for training, and the last 25% of the data was used for testing for **both linear and logistic regression**. Training values are the features, and testing values are the target values. The code is given below:

```
first_75 = math.ceil(0.75*len(df1))
data_train = df1[:first_75, :] #first 75% of the data
data_test = df1[first_75:, :] #last 25% of the data

x_train = data_train[:, :-1]
y_train = data_train[:, -1]
x_test = data_test[:, :-1]  # First 9 columns
y_test = data_test[:, -1]   # Last column

print(x_train.shape)
print(x_test.shape)
```

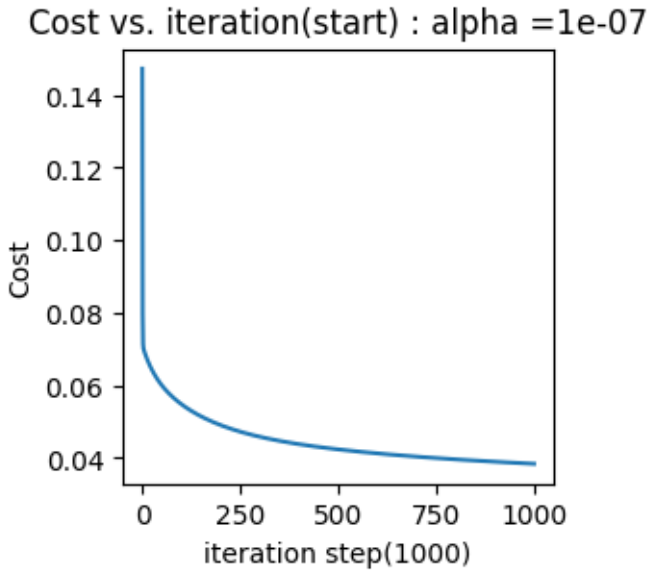
III. Experiment results

A. *Linear Regression Without Regularization*

- Before regularization, we tried to check out output for iteration value(1000) and $\alpha = 1.0e-7$

Alpha	Iteration	Cost
1.0e-7	100	0.15
	200	0.05
	⋮	⋮
	800	0.04
	900	0.04

α	Iterations	Target Value	Predicted Value
1.0e-7	1000	0.84	0.77
	1000	0.80	0.77
	1000	0.81	0.77



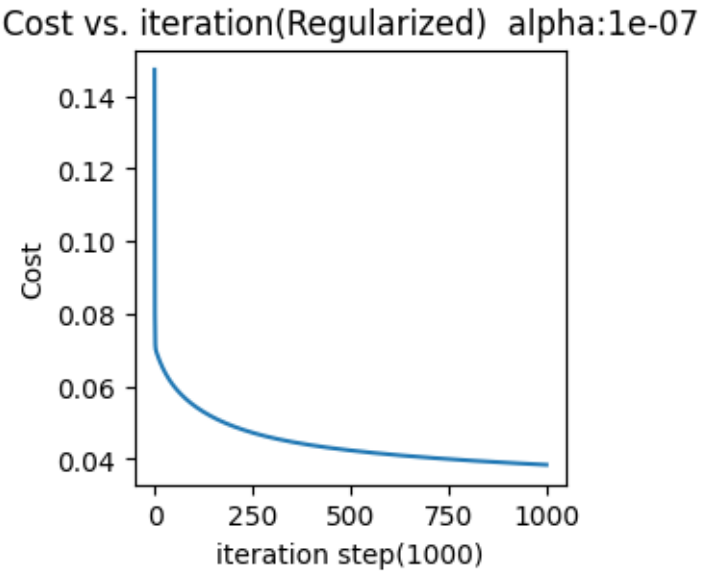
```
iterations = 1000 #3000
tmp_alpha = 1.0e-7 #1.0e-8,1.0e-9,1.0e-10

w_final, b_final, J_hist, p_hist
        = gradient_descent(x_train ,y_train, w_init, b_init,tmp_alpha,
            iterations,compute_cost, compute_gradient)

print(f"(w,b) found by gradient descent (alpha = {tmp_alpha:.1e}):
({w_final},
{b_final:8.4f})")

for i in range(5):
  print(f"alpha: {tmp_alpha:.1e}, target value: {y_test[i]:0.2f},
  predicted value: {np.dot(x_test[i], w_final) + b_final:0.2f}")

# plot cost versus iteration
fig, (ax1) = plt.subplots(1, 1, constrained_layout=True, figsize=(3,3))
ax1.plot(J_hist[:iterations])
ax1.set_title(f"Cost vs. iteration(start) : alpha ={tmp_alpha}");
ax1.set_ylabel('Cost')            ;
ax1.set_xlabel(f'iteration step({iterations})')  ;
plt.show()

print(x_train.shape)
print(x_test.shape)
```

B. Linear Regression with regularization

Alpha	Iteration	Cost
1.0e-7	100	0.15
	200	0.04
	⋮	⋮
	800	0.04

α	Iterations	Target Value	Predicted Value
1.0e-7	1000	0.84	0.76
	1000	0.80	0.77
	1000	0.81	0.77

After Regularization cost vs Iteration for ($\alpha = 1.0e-7$)



So, in conclusion, there is not so much change after performing regularization

C. *Logistic Regression without regularization*

Alpha	Lambda	Iteration	Cost	Train Accuracy (%)	Test Accuracy (%)
0.001	0	0 1000 2000 ⋮ 9000 9999	4.03 3.10 2.19 ⋮ 0.31 0.29	88.80	76.28

D. *Logistic Regression with regularization*

Alpha	Lambda	Iteration	Cost	Train Accuracy (%)	Test Accuracy (%)
0.001	0	0 1000 2000 ⋮ 9000 9999	4.03 3.10 2.15 ⋮ 0.30 0.24	88.87	77.05

```
#Compute accuracy on our training set
p = predict(X_train, w,b)
print('Train␣Accuracy:␣%f'%(np.mean(p == y_train) * 100))
p = predict(X_test, w,b)
print('Test␣Accuracy:␣%f'%(np.mean(p == y_test) * 100))
```

So, in conclusion, there is a little bit change after performing regularization

IV. Discussion

In this section, we discuss key aspects of our analysis and implementation.

A. *Data Analysis and Pre-processing*

We meticulously analyzed the dataset, employing data pre-processing techniques. This involved dropping unnecessary columns and regularization, which were crucial steps to ensure the quality and reliability of our analysis.

B. *Sigmoid, Cost, and Gradient Descent Implementation*

We have performed regularization on both linear and logistic regression models.

For linear regression, we have added the regularization term to the existing cost and gradient function and performed regularization. We have seen less change after regularization than the feature scaling we did in the previous assignment.

For logistic regression, we have worked on the given code. We have added the regularization term to the cost and gradient descent function. We have seen a slight change in accuracy after regularization.

C. *Effect of Alpha on Data*

Our exploration included observing the impact of alpha on the dataset. Notably, we varied the alpha value and observed corresponding changes in the data. For instance, when $\alpha = 0$, we observed a significant reduction in cost over the iterations. However, when alpha was increased to $\alpha = 200$, the rate of cost reduction was worse compared to the previous case. Furthermore, the optimisation process diverged when alpha was set to a very high value, such as $\alpha = 20000$.

These findings underscore the importance of thoughtful data preprocessing, algorithm implementation, and parameter tuning in the success of machine learning models.