20F-0292_20F-0297_BCS 7D_ML ASSIGNMENT 1

Group Assignment

QUESTION # 1:

Open in Colab

Download training dataset, which consists of the input X file and the corresponding output Y file. The input X file (DataX) has three attributes: the living area, the number of bedrooms, and the number of floors, while the output Y file (DataY) represents the house prices in response to these attributes. There are $m = 50$ training examples. Perform the following tasks,

In [278…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [308…

```
# Read 'DataX.dat' and 'DataY.dat' files with whitespace (space or tab) as delimiter, without assuming headers.
X = pd.read_csv('DataX.dat', sep='\s+', header=None)
Y = pd.read_csv('DataY.dat', sep='\s+', header=None)
```

In [309…

```
X.head(5)
```

Out[309…

	0	1	2
0	2104.0	3.0	2.0
1	1600.0	3.0	2.0
2	2400.0	3.0	2.0
3	1416.0	2.0	1.0
4	3000.0	4.0	3.0

In [310…

```
Y.head(5)
```

Out[310…

	0
0	399900.0
1	329900.0
2	369000.0
3	232000.0
4	539900.0

In [311…

```
X[3] = 1 # Add a third column with all values = 1
```

In [312…

```python
X.head(5)
```

Out[312…

	0	1	2	3
0	2104.0	3.0	2.0	1
1	1600.0	3.0	2.0	1
2	2400.0	3.0	2.0	1
3	1416.0	2.0	1.0	1
4	3000.0	4.0	3.0	1

In [313…

```python
X = X[[3, 0, 1, 2]]  # Rearranging the columns
```

In [314…

```python
X.columns = ['x_0', 'x_1', 'x_2', 'x_3']  # Renaming the columns
```

In [315…

```python
Y.columns = ['y_0'] # Renaming the columns
```

In [316…

```python
X.head(5)
```

Out[316…

	x_0	x_1	x_2	x_3
0	1	2104.0	3.0	2.0
1	1	1600.0	3.0	2.0
2	1	2400.0	3.0	2.0
3	1	1416.0	2.0	1.0
4	1	3000.0	4.0	3.0

In [317…

```python
Y.head(5)
```

Out[317…

	y_0
0	399900.0
1	329900.0
2	369000.0
3	232000.0
4	539900.0

In [318…

```python
# First both dataframes to numpy array/ feature matrix

X_train = X.values # For Case A
Y_train = Y.values # For Case A
```

In [319…

```python
print(f"Shape of X: {X_train.shape}")
print(f"Shape of Y: {Y_train.shape}")
```

```
Shape of X: (50, 4)
Shape of Y: (50, 1)
```

Case A:

(a) Gradient Descent

- Preprocess the data,
- Implement gradient descent algorithm with a learning rate = 0.02.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ldots, \theta_n)$$

⭐ Preprocessing Using feature scaling

We normalize

In [321…

```
for i in range(5):
    mean_x1 = np.mean(X_train[:, 1])
    std_x1 = np.std(X_train[:, 1])
    X_train[:, 1] = (X_train[:, 1] - mean_x1) / (std_x1)

for i in range(5):
    mean_x2 = np.mean(X_train[:, 2])
    std_x2 = np.std(X_train[:, 2])
    X_train[:, 2] = (X_train[:, 2] - mean_x2) / (std_x2)

for i in range(5):
    mean_x3 = np.mean(X_train[:, 3])
    std_x3 = np.std(X_train[:, 3])
    X_train[:, 3] = (X_train[:, 3] - mean_x3) / (std_x3)
```

In [322…

```
print(f"Values of normalized x_0 are: {np.min(X_train[0])} <= x_0 <= {np.max(X_train[0])}")
print(f"Values of normalized x_1 are: {np.min(X_train[1])} <= x_1 <= {np.max(X_train[1])}")
print(f"Values of normalized x_2 are: {np.min(X_train[2])} <= x_2 <= {np.max(X_train[2])}")
print(f"Values of normalized x_3 are: {np.min(X_train[3])} <= x_3 <= {np.max(X_train[3])}")
```

```
Values of normalized x_0 are: -0.24854790640047997 <= x_0 <= 1.0
Values of normalized x_1 are: -0.5044594772666824 <= x_1 <= 1.0
Values of normalized x_2 are: -0.24854790640047997 <= x_2 <= 1.0
Values of normalized x_3 are: -1.8019723214034802 <= x_3 <= 1.0
```

In [323…

```
# Reshaping

X_train = X_train.T
Y_train = Y_train.reshape(1, X_train.shape[1])

print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of Y_train: {Y_train.shape}")
```

```
Shape of X_train: (4, 50)
Shape of Y_train: (1, 50)
```

⭐ Implementing Gradient Descent

Formulas

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Repeat {

$$\frac{\partial}{\partial\theta_j} J(\theta)$$

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

(simultaneously update θ_j for $j = 0, \ldots, n$) }

In [324…

```
X_train.shape
```

Out[324…

```
(4, 50)
```

In [325…

```
def training_model(X, y, alpha, total_iterations):
    m = X.shape[1]
    n = X.shape[0]

    cost_list = []
    # Initialize theta with zeros
    theta = np.zeros((n, 1))

    for i in range(total_iterations):
        predicted = np.dot(theta.T, X)

        cost = (1 / (2 * m)) * np.sum((predicted - y) ** 2) # Calculate the cost using MSE

        # Compute the gradient
        gradient = (1 / m) * np.dot(X, (predicted - y).T)

        theta = theta - alpha * gradient # Update the parameters

        cost_list.append(cost)

        if i % (total_iterations // 10) == 0:
            print(f"Cost after {i} iterations is {cost}")

    return theta, cost_list
```

In [326…

```
iterations = 60000
theta, costs = training_model(X_train, Y_train, 0.02, iterations)
```

```
Cost after 0 iterations is 63703357920.07
Cost after 6000 iterations is 2168815479.4035683
Cost after 12000 iterations is 2168795542.024405
Cost after 18000 iterations is 2168795531.720504
Cost after 24000 iterations is 2168795531.7151785
Cost after 30000 iterations is 2168795531.7151756
Cost after 36000 iterations is 2168795531.7151756
Cost after 42000 iterations is 2168795531.715175
Cost after 48000 iterations is 2168795531.7151747
Cost after 54000 iterations is 2168795531.7151747
```

In [327…

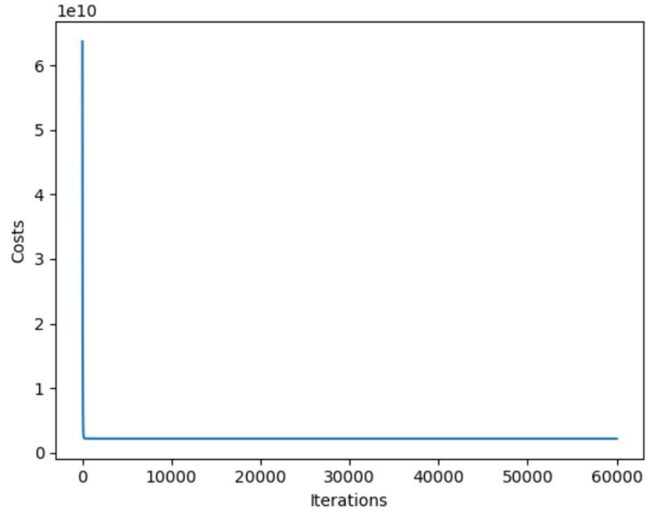
```
print("Values of parameters are:"); print(theta)
```

```
Values of parameters are:
[[335373.9       ]
 [100780.43797099]
 [ 38772.16952469]
 [-40555.622871  ]]
```

Plotting

In [339…

```
plt.plot(np.arange(iterations), costs)
plt.xlabel('Iterations')
plt.ylabel('Costs')
plt.show()
```



Case B:

(b) Consider the closed-form solution to a least square fit given as under. Implement it in order to calculate the values of the parameters for the same dataset.

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

In [266…

```
# Make feature matrix
x = X.values
y = Y.values
```

In [267…
```
a = np.dot(x.T, x) # Implement (X^T.X) of formula
```

In [268…
```
a_inv = np.linalg.pinv(a) # Implement (X^T.X)^-1 of formula
```

In [269…
```
b = np.dot(x.T, y) # Implement (X^T.y) of formula
```

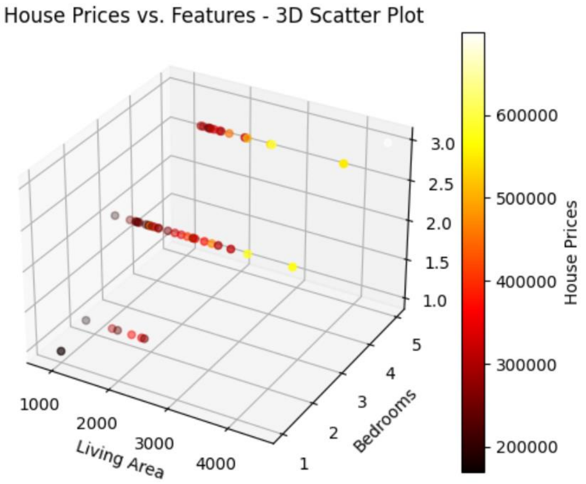
In [271…
```
theta_norm = np.dot(a, b) # Implement (X^T.X)^-1.(X^T.y) of formula i.e. whole formula
```

In [272…
```
print("Closed form solution is:"); print(theta_norm)
```

```
Closed form solution is:
[[3.75230868e+15]
 [8.66955207e+18]
 [1.24429786e+16]
 [8.56005900e+15]]
```

Case C:

In [340…
```
from mpl_toolkits.mplot3d import Axes3D

X = np.loadtxt("DataX.dat")
Y = np.loadtxt("DataY.dat")

# Extract individual features
living_area = X[:, 0]
bedrooms = X[:, 1]
floors = X[:, 2]

# Create a 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Scatter plot with color gradient based on house prices
sc = ax.scatter(living_area, bedrooms, floors, c=Y, cmap=plt.hot())

# Customize the axes labels
ax.set_xlabel('Living Area')
ax.set_ylabel('Bedrooms')
ax.set_zlabel('Floors')
ax.set_title('House Prices vs. Features - 3D Scatter Plot')

# Add a color bar to show the price range
cbar = plt.colorbar(sc)
cbar.set_label('House Prices')

plt.show()
```



Case D:

(d) Make comparison of both results obtained in case of (a) and (b).

- Case A:

Values of parameters:

- Intercept (θ_0): 335,373.9
- Coefficient for Living Area (θ_1): 100,780.44
- Coefficient for Bedrooms (θ_2): 38,772.17
- Coefficient for Floors (θ_3): -40,555.62

- Case B:

Closed form solution:

These parameter values are extremely large, e.g., $\theta_0 \approx 3.75e+15$, $\theta_1 \approx 8.67e+18$, $\theta_2 \approx 1.24e+16$, $\theta_3 \approx 8.56e+15$.

Comparison:

Scale of Parameter Values:

- Case A:

The parameter values are in a reasonable range and can be interpreted directly. For example, the coefficient for the Living Area (θ_1) suggests that, on average, an increase of 1 unit in the living area results in an increase of approximately $100,780.44 in house price.

- Case B:

The parameter values in Case B are exceptionally large and likely indicate a problem with the model or numerical instability. It's not practical to interpret such large parameter values in the context of a linear regression model.

Interpretability:

- **Case A** provides interpretable coefficients that can be used to understand the relationships between the input features (Living Area, Bedrooms, and Floors) and the output (House Prices).

- **Case B** Parameter values are so large that they do not provide meaningful insights or interpretations.

Potential Issues:

- **Case B** Parameter values suggest numerical instability or problems with the modeling process. Such large parameter values are typically indicative of issues like multicollinearity, improper feature scaling, or divergence in the optimization algorithm.

QUESTION # 2:

Question NO#2

Prove that the derivative of logistic (Sigmoid) function:-

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \text{is} \quad \frac{d\sigma(x)}{dx} = \sigma(x)\cdot(1-\sigma(x))$$

Solution:-

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$$

$$= \frac{d}{dx}(1+e^{-x})^{-1}$$

Chain Rule

$$= (-1)(1+e^{-x})^{-2}\cdot\frac{d}{dx}(1+e^{-x})$$

$$= -(1+e^{-x})^{-2}\cdot\left[\frac{d}{dx}(1) + \frac{d}{dx}(e^{-x})\right]$$

$$= -(1+e^{-x})^{-2}\cdot\left[0 + e^{-x}\frac{d}{dx}(-x)\right]$$

$$= -(1+e^{-x})^{-2}(-e^{-x})$$

$$= \frac{e^{-x}}{(1+e^{-x})^{2}}$$

$$= \frac{e^{-x}}{(1+e^{-x})}\cdot\frac{1}{(1+e^{-x})}$$

$$= \left(\frac{-1}{1+e^{x}} + 1\right)\cdot\frac{1}{(1+e^{-x})}$$

$$= \left(1 - \frac{1}{1+e^{-x}}\right)\cdot\left(\frac{1}{1+e^{-x}}\right)$$

$$\frac{d\,\sigma(x)}{dx} = (1-\sigma(x)) \cdot \sigma(x)$$

Hence Proved

Tangent Function

$\tanh(x) = \dfrac{e^x + e^{-x}}{e^x - e^{-x}}$ is $\tanh'(x) = 1 - \tanh^2(x)$

$$\frac{d}{dx}\tanh(x) = \frac{d}{dx}\left(\frac{e^x + e^{-x}}{e^x - e^{-x}}\right)$$

So taking Derivative on Both Side

$$= \frac{(e^x + e^{-x})\frac{d}{dx}(e^x - e^{-x}) - (e^x - e^{-x})\frac{d}{dx}(e^x + e^{-x})}{(e^x + e^{-x})^2}$$

$$= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2}$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

As we know $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

Hence Proved

QUESTION # 3:

Open in Colab

Problem Statement:

Implement the logistic regression in order to classify the houses into two classes, "Costly" and "Not Costly", using the same input data "DataX" and the classes in "ClassY" file provided in Question 1. The "ClassY" file contains two values: **1** for "Costly" class and **0** for "Not Costly" class.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:
```
# Read 'DataX.dat' and 'DataY.dat' files with whitespace (space or tab) as delimiter, without assuming headers.
X = pd.read_csv('DataX.dat', sep='\s+', header=None)
Y = pd.read_csv('ClassY.dat')
```

In [4]:
```
X.head()
```

Out[4]:

	0	1	2
0	2104.0	3.0	2.0
1	1600.0	3.0	2.0
2	2400.0	3.0	2.0
3	1416.0	2.0	1.0
4	3000.0	4.0	3.0

In [5]:
```
Y.head()
```

Out[5]:

	1
0	1
1	1
2	0
3	1
4	0

In [6]:

```
print("Total Training examples of X are: ", len(X))
print("Total Training examples of Y are: ", len(Y))
```

```
Total Training examples of X are:  50
Total Training examples of Y are:  49
```

In [7]:

```
# Remove the last row (last sample) from the X DataFrame
X = X.iloc[:-1]
```

In [8]:

```
print("Total Training examples of X are: ", len(X))
print("Total Training examples of Y are: ", len(Y))
```

```
Total Training examples of X are:  49
Total Training examples of Y are:  49
```

In [9]:

```
X[3] = 1 # Add a third column with all values = 1
```

In [10]:

```
X = X[[3, 0, 1, 2]]  # Rearranging the columns
```

In [11]:

```
X.columns = ['x_0', 'x_1', 'x_2', 'x_3']  # Renaming the columns
```

In [12]:

```
Y.columns = ['y_0'] # Renaming the columns
```

In [13]:

```
print(X.columns)
print(Y.columns)
print("--------------------------------")
print(X.head(5))
print("--------------------------------")
print(Y.head(5))
print("--------------------------------")
```

```
Index(['x_0', 'x_1', 'x_2', 'x_3'], dtype='object')
Index(['y_0'], dtype='object')
--------------------------------
   x_0     x_1  x_2  x_3
0    1  2104.0  3.0  2.0
1    1  1600.0  3.0  2.0
2    1  2400.0  3.0  2.0
3    1  1416.0  2.0  1.0
4    1  3000.0  4.0  3.0
--------------------------------
   y_0
0    1
1    1
2    0
3    1
4    0
--------------------------------
```

In [14]:

```
print(f"Shape of X: {X.shape}")
print(f"Shape of Y: {Y.shape}")
```

```
Shape of X: (49, 4)
Shape of Y: (49, 1)
```

In [15]:

```
# Changing to vectors/feature matrix

X_train = X.values
Y_train = Y.values
```

Feature Scaling

In [17]:
```
for i in range(5):
    mean_x1 = np.mean(X_train[:, 1])
    std_x1 = np.std(X_train[:, 1])
    X_train[:, 1] = (X_train[:, 1] - mean_x1) / (std_x1)

for i in range(5):
    mean_x2 = np.mean(X_train[:, 2])
    std_x2 = np.std(X_train[:, 2])
    X_train[:, 2] = (X_train[:, 2] - mean_x2) / (std_x2)

for i in range(5):
    mean_x3 = np.mean(X_train[:, 3])
    std_x3 = np.std(X_train[:, 3])
    X_train[:, 3] = (X_train[:, 3] - mean_x3) / (std_x3)
```

In [18]:
```
print(f"Values of normalized x_0 are: {np.min(X_train[0])} <= x_0 <= {np.max(X_train[0])}")
print(f"Values of normalized x_1 are: {np.min(X_train[1])} <= x_1 <= {np.max(X_train[1])}")
print(f"Values of normalized x_2 are: {np.min(X_train[2])} <= x_2 <= {np.max(X_train[2])}")
print(f"Values of normalized x_3 are: {np.min(X_train[3])} <= x_3 <= {np.max(X_train[3])}")
```

```
Values of normalized x_0 are: -0.25122971720853104 <= x_0 <= 1.0
Values of normalized x_1 are: -0.4944935530779656 <= x_1 <= 1.0
Values of normalized x_2 are: -0.25122971720853104 <= x_2 <= 1.0
Values of normalized x_3 are: -1.790011735110784 <= x_3 <= 1.0
```

In [19]:
```
# Reshaping

X_train = X_train.T
Y_train = Y_train.reshape(1, X_train.shape[1])

print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of Y_train: {Y_train.shape}")
```

```
Shape of X_train: (4, 49)
Shape of Y_train: (1, 49)
```

In [20]:
```
# First we implement the sigmoid function

def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

In [21]:
```
def training_model(X, y, alpha, total_iterations):
    m = X.shape[1]
    n = X.shape[0]

    # Initialize theta with zeros
    theta = np.zeros((n, 1))

    cost_list = []
    for i in range(total_iterations):
        Z = np.dot(theta.T, X)
        predicted = sigmoid(Z)

        cost = (-1/m) * np.sum(y*np.log(predicted) + (1-y)*np.log(1-predicted))

        # Compute the gradient
        gradient = (1/m) * np.dot(X, (predicted - y).T)

        theta = theta - alpha * gradient

        cost_list.append(cost)

        if i % (total_iterations // 10) == 0:
            print(f"Cost after {i} iterations is {cost}")

    return theta, cost_list
```

In [22]:
```
iterations = 10000000
theta, costs = training_model(X_train, Y_train, 0.00002, iterations)
```

```
Cost after 0 iterations is 0.6931471805599453
Cost after 1000000 iterations is 0.6716537536718094
Cost after 2000000 iterations is 0.6650165434368666
Cost after 3000000 iterations is 0.6601006730235618
Cost after 4000000 iterations is 0.6564083100736954
Cost after 5000000 iterations is 0.6535950954771987
Cost after 6000000 iterations is 0.6514180442369641
Cost after 7000000 iterations is 0.6497065648189989
Cost after 8000000 iterations is 0.6483404434276148
Cost after 9000000 iterations is 0.6472342152343314
```

Plotting

In [23]:

```
plt.plot(np.arange(iterations), costs)
plt.xlabel('Iterations')
plt.ylabel('Parameters')
plt.show()

## Here on the y axis if cost and not the parameters. Correction
```