NAME: SYED MUZZAMIL WASEEM

SID: 11067

CID: 113083

ASSIGNMENT # 01 (MACHINE LEARNING)

1- Derive equations for Gradient Descent by using Linear Regression cost function.

ANSWER 01

In linear regression, the cost function is:

$$J = \frac{1}{n} \sum_{i=1}^{n} [Y_i - (mx_i - c)]^2$$

Partial Derivative

m:

$$\frac{\partial J}{\partial m} = \frac{2}{n} \sum_{n=1}^{n} (-x_i) [Y_i - (mx_i + c)]$$

$$\because Y_i - mx_i - c \Rightarrow \frac{\partial m}{\partial m} = 1$$

$$0 - \sum_{i=1}^{n} - x_i y_i + m x_i^2 + x_i c$$

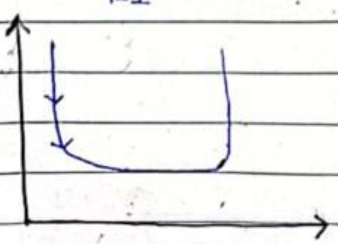
$$\sum_{i=1}^{n} m x_i^2 + \sum_{i=1}^{n} x_i c - \sum_{i=1}^{n} x_i Y_i$$

c:

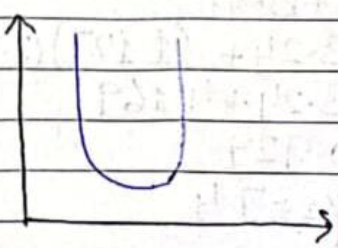
$$\frac{\partial J}{\partial c} = \frac{2}{n} \sum_{i=1}^{n} - [Y_i - (mx_i + c)]$$

$$\because Y_i - mx_i - c = \frac{\partial c}{\partial c} = 1$$

$$0 = \sum_{i=1}^{n} - Y_i + mx_i - c$$



$LR = \lambda_1 x$



Date_____ 20___

Incorporating our model for finite no of steps because of differentiation so, we say

$$m = m - \lambda \frac{\partial J}{\partial m}$$

$$c = c - \lambda \frac{\partial J}{\partial c} \qquad \lambda = LR$$

In terms of Q:

$Y - n(x, Q^n)$

$\hat{Y} - n(x, Q) + \epsilon$

So,

$\hat{Y} = Q_0 + Q_1 x + Q_2 x^2 + Q_3 x^3 \ldots\ldots$

$\hat{Y} = Q_n x^n$

$$J(x, Q, y) = \frac{1}{\partial m} \sum_{i=1}^{m} (\hat{Y}_i - Y_i)^2$$

$$Q^+ = Q^- - x \hat{\Delta} J$$

Differentiate w.r.t. Q

$$\frac{\partial J}{\partial Q} = \frac{1}{\partial m} \sum_{i=1}^{m} \partial(\hat{Y}_i - Y_i) \frac{Q}{\partial Q} (Q^T \bar{x}_j - Y_i)$$

$$Y_i = Q^T \bar{x}_i$$

$$\Rightarrow \Delta J = \frac{1}{m} \sum_{i=1}^{m} (\hat{Y}_i - Y_i) \bar{x}_i$$

2- Follow above steps and run the above code to get optimized parameters for Linear Regression by using Gradient Descent. Print the optimized parameters and visualizations and attach in your file.

LINEAR REGRESSION BY USING GRADIENT DESCENT

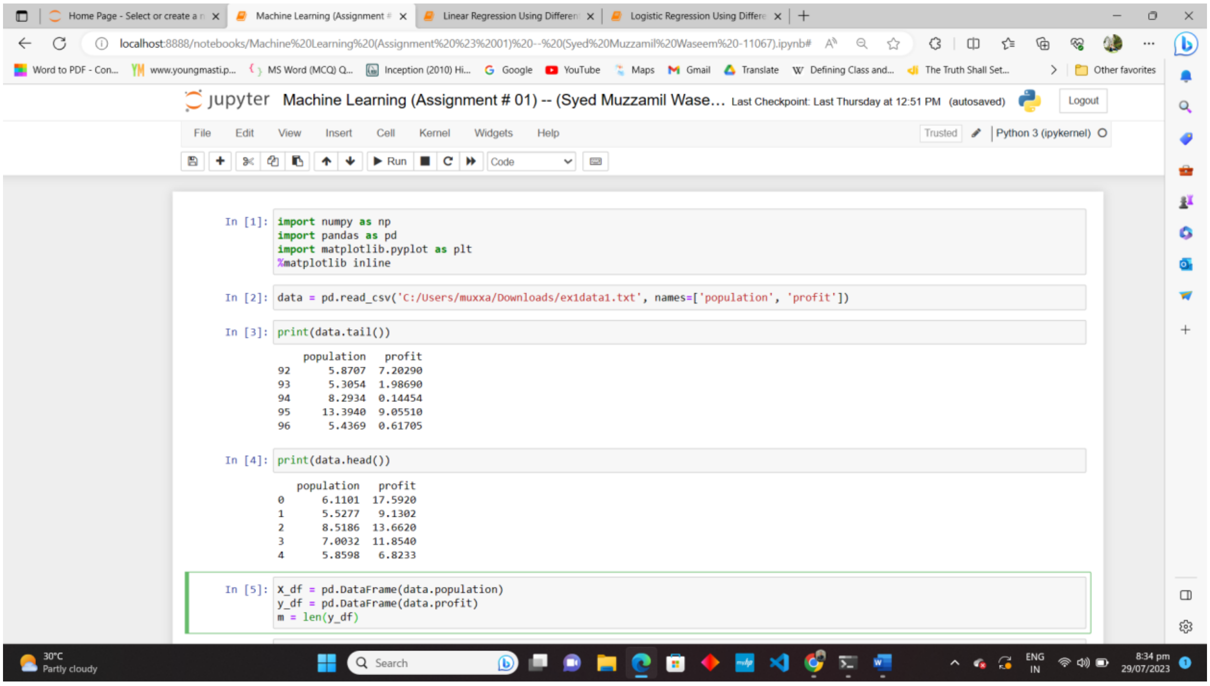
CODE:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

data = pd.read_csv('C:/Users/muxxa/Downloads/ex1data1.txt', names=['population', 'profit'])
```

```
print(data.tail())
print(data.head())
X_df = pd.DataFrame(data.population)
y_df = pd.DataFrame(data.profit)
m = len(y_df)
plt.figure(figsize=(10, 8))
plt.plot(X_df.values, y_df.values, 'kx')
plt.xlabel('Population of City in 10,000s')
plt.ylabel('Profit in $10,000s')
plt.title('Profit vs. Population')
plt.show()
iterations = 1000
alpha = 0.01
X_df['intercept'] = 1
X = np.array(X_df)
y = np.array(y_df).flatten()
theta = np.array([0, 0])
def cost_function(X, y, theta):
    m = len(y)
    hypothesis = X.dot(theta)
    J = np.sum((hypothesis - y) ** 2) / 2 / m
    return J
print("Cost before Gradient Descent:", cost_function(X, y, theta))
print(X)
def gradient_descent(X, y, theta, alpha, iterations):
    cost_history = [0] * iterations

    for iteration in range(iterations):
        hypothesis = X.dot(theta)
        loss = hypothesis - y
        gradient = X.T.dot(loss) / m
```

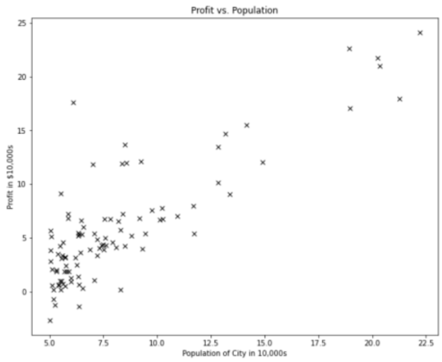
```
        theta = theta - alpha * gradient
        cost = cost_function(X, y, theta)
        cost_history[iteration] = cost
    return theta, cost_history
gd = gradient_descent(X, y, theta, alpha, iterations)
print("Optimized Parameters (theta):", gd[0])
best_fit_x = np.linspace(0, 25, 20)
best_fit_y = gd[0][1] + gd[0][0] * best_fit_x
plt.figure(figsize=(10, 6))
plt.plot(X_df['population'], y_df['profit'], 'kx')
plt.plot(best_fit_x, best_fit_y, '-')
plt.axis([0, 25, -5, 25])
plt.xlabel('Population of City in 10,000s')
plt.ylabel('Profit in $10,000s')
plt.title('Profit vs. Population with Linear Regression Line')
plt.show()
```

CODE + OUTPUTS:



Jupyter Machine Learning (Assignment # 01) -- (Syed Muzzamil Wase... Last Checkpoint: Last Thursday at 12:51 PM (autosaved)

In [7]:
```python
plt.figure(figsize=(10, 8))
plt.plot(X_df.values, y_df.values, 'kx')
plt.xlabel('Population of City in 10,000s')
plt.ylabel('Profit in $10,000s')
plt.title('Profit vs. Population')
plt.show()
```



Jupyter Machine Learning (Assignment # 01) -- (Syed Muzzamil Wase... Last Checkpoint: Last Thursday at 12:51 PM (autosaved)

In [8]:
```python
iterations = 1000
alpha = 0.01
```

In [9]:
```python
X_df['intercept'] = 1
```

In [10]:
```python
X = np.array(X_df)
y = np.array(y_df).flatten()
theta = np.array([0, 0])
```

In [11]:
```python
def cost_function(X, y, theta):
    m = len(y)
    hypothesis = X.dot(theta)
    J = np.sum((hypothesis - y) ** 2) / 2 / m
    return J
```

In [12]:
```python
print("Cost before Gradient Descent:", cost_function(X, y, theta))
```

```
Cost before Gradient Descent: 32.072733877455676
```

In [14]:
```python
print(X)
```

```
[[ 6.1101  1.    ]
 [ 5.5277  1.    ]
 [ 8.5186  1.    ]
 [ 7.0032  1.    ]
 [ 5.8598  1.    ]
 [ 8.3829  1.    ]
 [ 7.4764  1.    ]
 [ 8.5781  1.    ]
 [ 6.4862  1.    ]
 [ 5.0546  1.    ]
 [ 5.7107  1.    ]
 [14.164   1.    ]
 [ 5.734   1.    ]
 [ 8.4084  1.    ]
 [ 5.6407  1.    ]
```

jupyter Machine Learning (Assignment # 01) -- (Syed Muzzamil Wase... Last Checkpoint: Last Thursday at 12:51 PM (autosaved)

```
 [ 5.7292  1.    ]
 [ 5.1884  1.    ]
 [ 6.3557  1.    ]
 [ 9.7687  1.    ]
 [ 6.5159  1.    ]
 [ 8.5172  1.    ]
 [ 9.1802  1.    ]
 [ 6.002   1.    ]
 [ 5.5204  1.    ]
 [ 5.0594  1.    ]
 [ 5.7077  1.    ]
 [ 7.6366  1.    ]
 [ 5.8707  1.    ]
 [ 5.3054  1.    ]
 [ 8.2934  1.    ]
 [13.394   1.    ]
 [ 5.4369  1.    ]]
```

In [15]:

```python
def gradient_descent(X, y, theta, alpha, iterations):
    cost_history = [0] * iterations

    for iteration in range(iterations):
        hypothesis = X.dot(theta)
        loss = hypothesis - y
        gradient = X.T.dot(loss) / m
        theta = theta - alpha * gradient
        cost = cost_function(X, y, theta)
        cost_history[iteration] = cost

    return theta, cost_history
```

In [16]:

```python
gd = gradient_descent(X, y, theta, alpha, iterations)
```

In [17]:

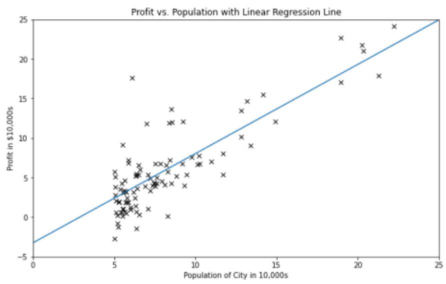
```python
print("Optimized Parameters (theta):", gd[0])
```

```
Optimized Parameters (theta): [ 1.1272942  -3.24140214]
```

jupyter Machine Learning (Assignment # 01) -- (Syed Muzzamil Wase... Last Checkpoint: Last Thursday at 12:51 PM (autosaved)

In [24]:

```python
best_fit_x = np.linspace(0, 25, 20)
best_fit_y = gd[0][1] + gd[0][0] * best_fit_x

plt.figure(figsize=(10, 6))
plt.plot(X_df['population'], y_df['profit'], 'kx')
plt.plot(best_fit_x, best_fit_y, '-')
plt.axis([0, 25, -5, 25])
plt.xlabel('Population of City in 10,000s')
plt.ylabel('Profit in $10,000s')
plt.title('Profit vs. Population with Linear Regression Line')
plt.show()
```



In [ ]:

3- By using the parameters obtained in above question, manually solve linear regression hypothesis equation for x=3.7 and x=7.4 by showing all necessary steps.

ANSWER 03:

Given that

$x_1 = 3.7$ $\quad$ $b_1 = 1.127$

$x_2 = 7.4$ $\quad$ $b_0 = 3.24$

For $x_1 = 3.7$

$$Y_1 = b_0 + b_1 x_1$$
$$= -3.24 + (1.127)(3.7)$$
$$= -3.24 + 4.169$$
$$Y_1 = 0.929$$

For $x_2 = 7.4$

$$Y_2 = b_0 + b_1 x_2$$
$$= -3.24 + (1.127)(7.4)$$
$$= -3.24 + 8.3398$$
$$Y_2 = 5.0998$$

4- Search a dataset for Linear Regression and apply same algorithm on your dataset. Print the optimized parameters and visualizations and attach in your file. Also attach the code of this part in your file.

i- LOGISTIC REGRESSION USING DIFFERENT DATASET:

Reason for using Different dataset: The dataset provided appears to have two columns, which could be potential features, but there is no indication of a binary target variable. This dataset is downloaded from Kaggle and dataset file is attached in Zip file.

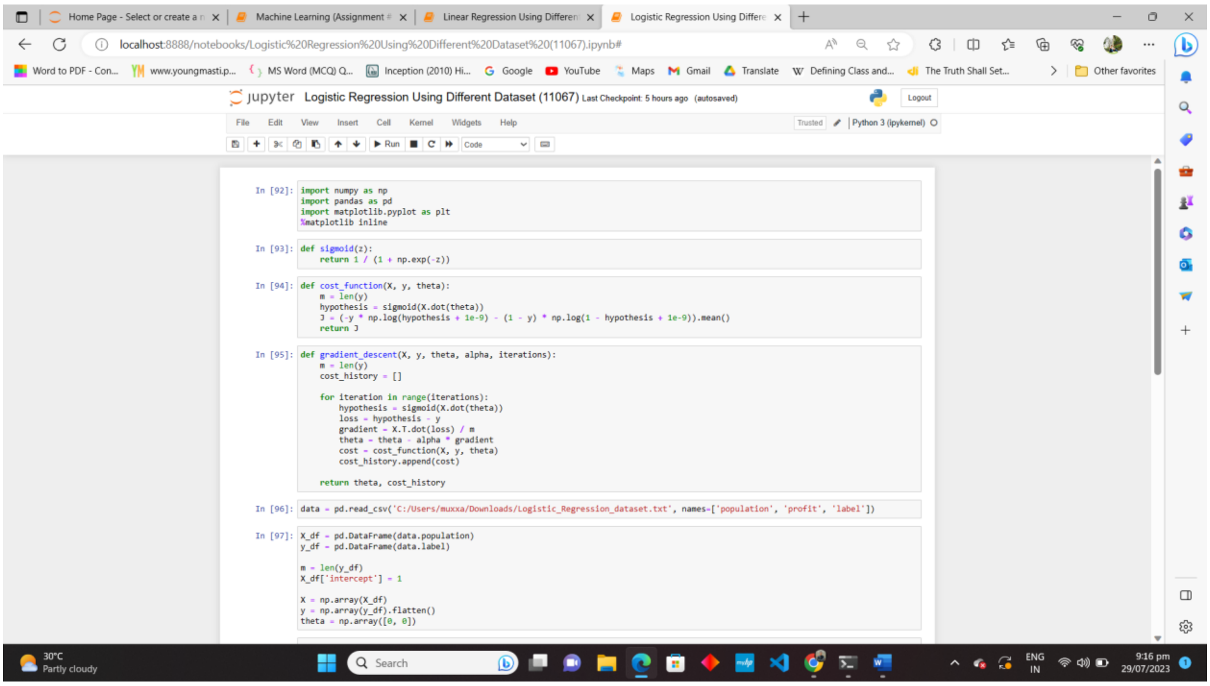
CODE:

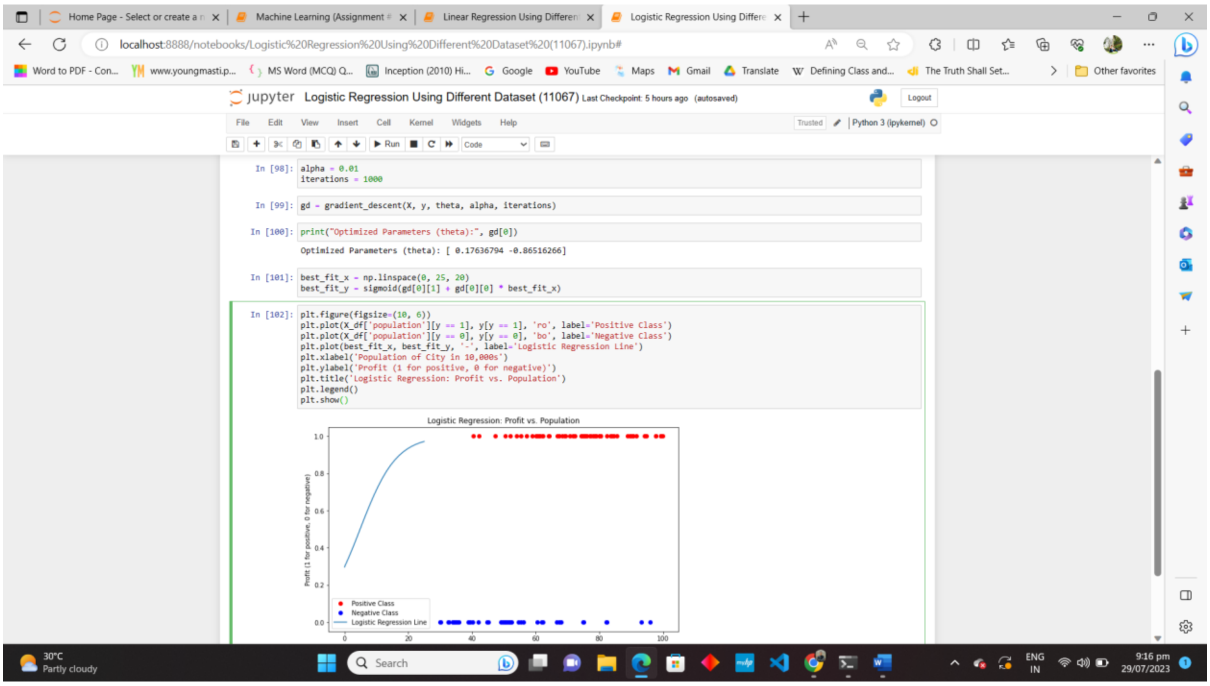
```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt
```

```
%matplotlib inline
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
def cost_function(X, y, theta):
    m = len(y)
    hypothesis = sigmoid(X.dot(theta))
    J = (-y * np.log(hypothesis + 1e-9) - (1 - y) * np.log(1 - hypothesis + 1e-9)).mean()
    return J
def gradient_descent(X, y, theta, alpha, iterations):
    m = len(y)
    cost_history = []

    for iteration in range(iterations):
        hypothesis = sigmoid(X.dot(theta))
        loss = hypothesis - y
        gradient = X.T.dot(loss) / m
        theta = theta - alpha * gradient
        cost = cost_function(X, y, theta)
        cost_history.append(cost)

    return theta, cost_history
data = pd.read_csv('C:/Users/muxxa/Downloads/Logistic_Regression_dataset.txt',
names=['population', 'profit', 'label'])
X_df = pd.DataFrame(data.population)
y_df = pd.DataFrame(data.label)
m = len(y_df)
X_df['intercept'] = 1
X = np.array(X_df)
y = np.array(y_df).flatten()
theta = np.array([0, 0])
alpha = 0.01
```

```
iterations = 1000

gd = gradient_descent(X, y, theta, alpha, iterations)

print("Optimized Parameters (theta):", gd[0])

best_fit_x = np.linspace(0, 25, 20)

best_fit_y = sigmoid(gd[0][1] + gd[0][0] * best_fit_x)

plt.figure(figsize=(10, 6))

plt.plot(X_df['population'][y == 1], y[y == 1], 'ro', label='Positive Class')

plt.plot(X_df['population'][y == 0], y[y == 0], 'bo', label='Negative Class')

plt.plot(best_fit_x, best_fit_y, '-', label='Logistic Regression Line')

plt.xlabel('Population of City in 10,000s')

plt.ylabel('Profit (1 for positive, 0 for negative)')

plt.title('Logistic Regression: Profit vs. Population')

plt.legend()

plt.show()
```

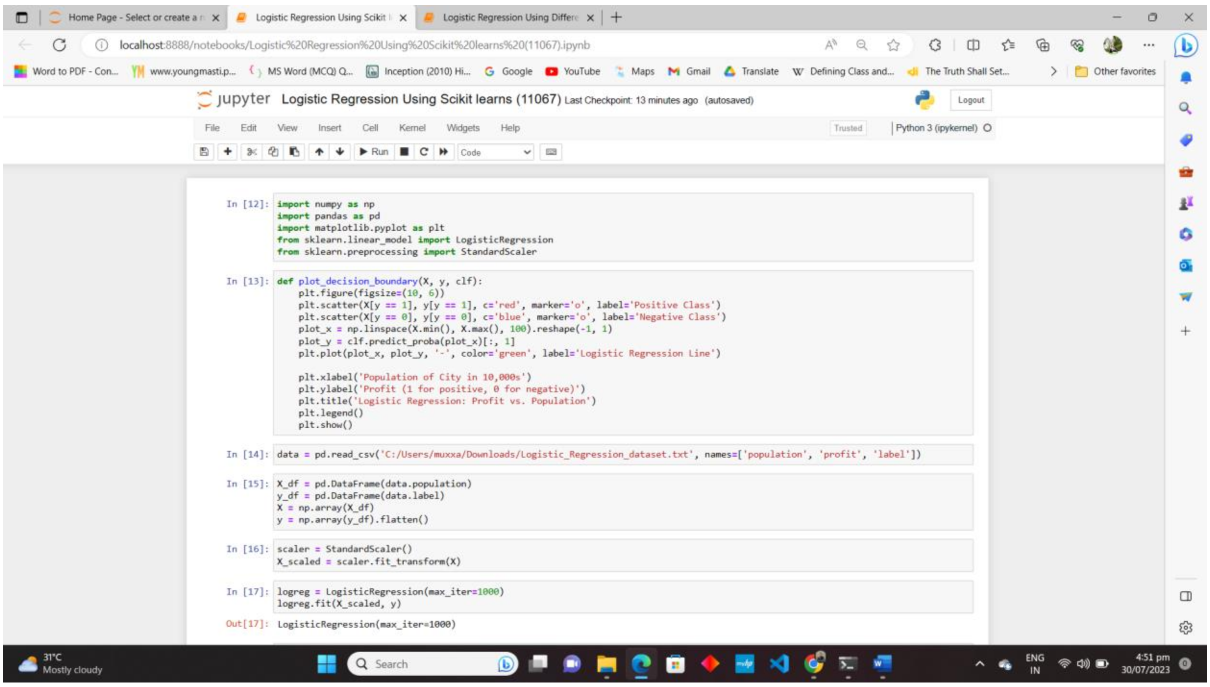
CODE + OUTPUTS:

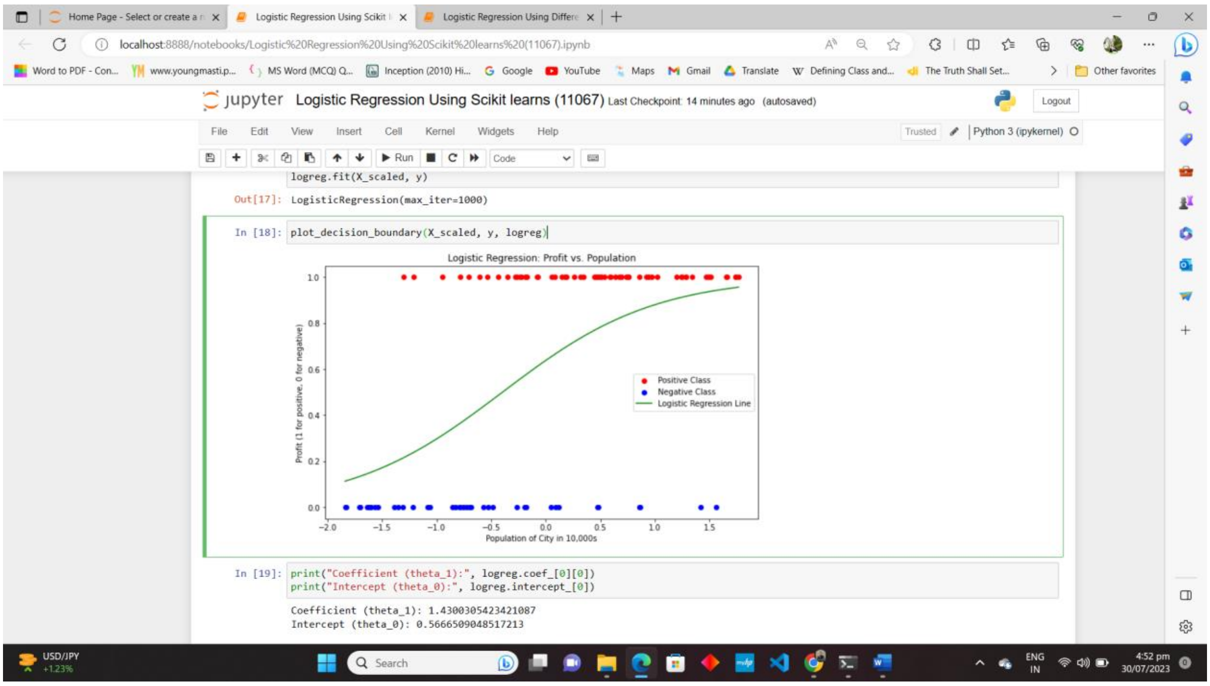




In Above Output Regression Line isn't closest to its datapoints in this modified code, we utilize scikit-learns Logistic Regression class, which handles feature scaling. We also plot the decision boundary based on the trained logistic regression model directly, which avoids the need for custom gradient descent. This should result in a better fit of the logistic regression line to the data points.

CODE+OUTPUT:





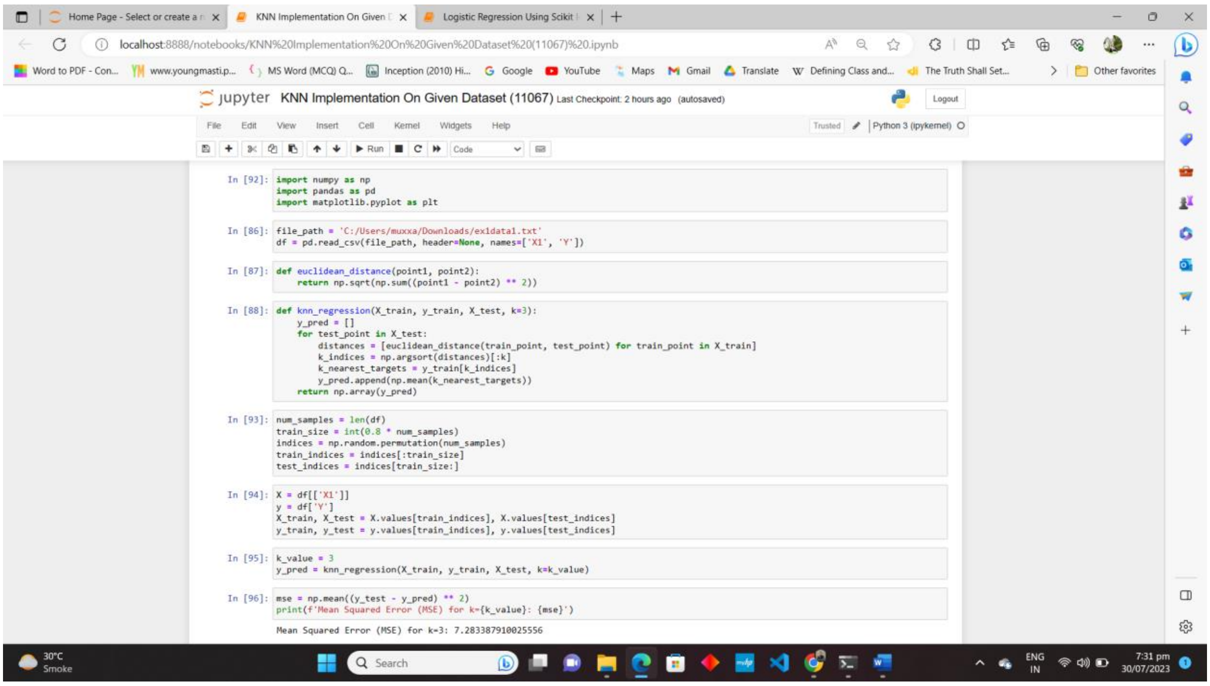
5- Implement KNN for the given Dataset.

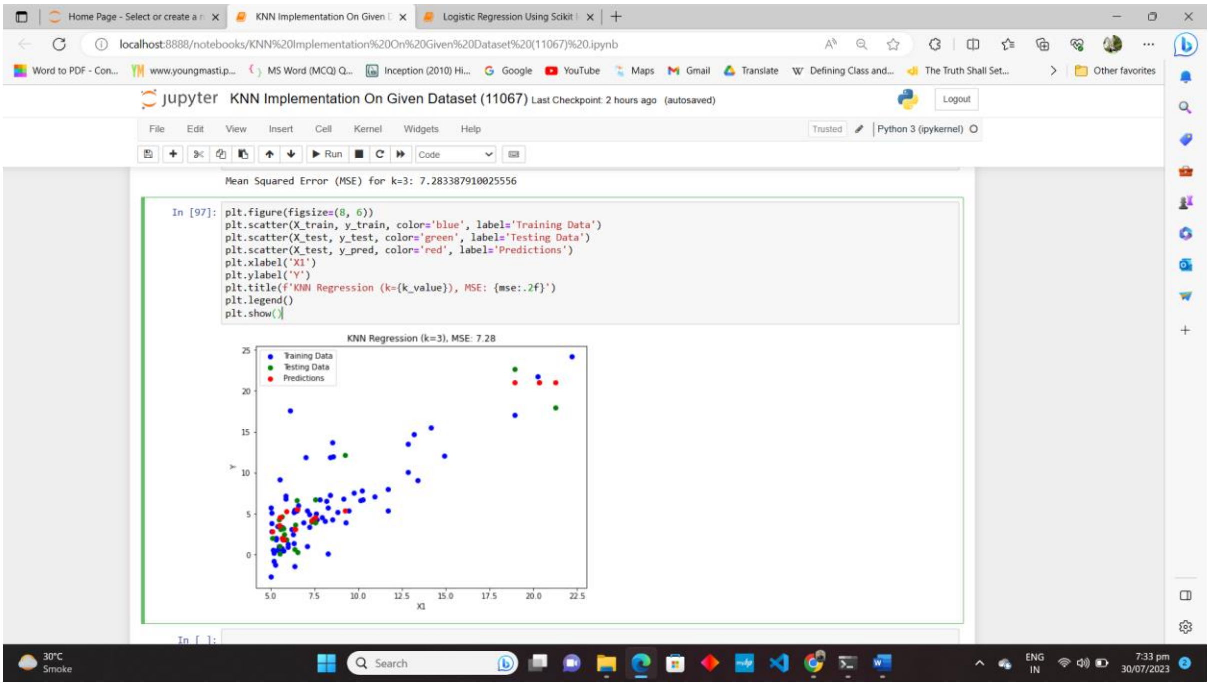
CODE:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

file_path = 'C:/Users/muxxa/Downloads/ex1data1.txt'

df = pd.read_csv(file_path, header=None, names=['X1', 'Y'])

def euclidean_distance(point1, point2):

    return np.sqrt(np.sum((point1 - point2) ** 2))

def knn_regression(X_train, y_train, X_test, k=3):

    y_pred = []

    for test_point in X_test:

        distances = [euclidean_distance(train_point, test_point) for train_point in X_train]

        k_indices = np.argsort(distances)[:k]

        k_nearest_targets = y_train[k_indices]

        y_pred.append(np.mean(k_nearest_targets))

    return np.array(y_pred)
```

```
num_samples = len(df)

train_size = int(0.8 * num_samples)

indices = np.random.permutation(num_samples)

train_indices = indices[:train_size]

test_indices = indices[train_size:]

X = df[['X1']]

y = df['Y']

X_train, X_test = X.values[train_indices], X.values[test_indices]

y_train, y_test = y.values[train_indices], y.values[test_indices]

k_value = 3

y_pred = knn_regression(X_train, y_train, X_test, k=k_value)

mse = np.mean((y_test - y_pred) ** 2)

print(f'Mean Squared Error (MSE) for k={k_value}: {mse}')

plt.figure(figsize=(8, 6))

plt.scatter(X_train, y_train, color='blue', label='Training Data')

plt.scatter(X_test, y_test, color='green', label='Testing Data')

plt.scatter(X_test, y_pred, color='red', label='Predictions')

plt.xlabel('X1')

plt.ylabel('Y')

plt.title(f'KNN Regression (k={k_value}), MSE: {mse:.2f}')

plt.legend()

plt.show()
```

CODE+OUTPUT:


KNN Implementation On Given Dataset (11067) Last Checkpoint: 2 hours ago (autosaved)
In [92]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
In [86]: file_path = 'C:/Users/muxxa/Downloads/ex1data1.txt'
df = pd.read_csv(file_path, header=None, names=['X1', 'Y'])
In [87]: def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))
In [88]: def knn_regression(X_train, y_train, X_test, k=3):
    y_pred = []
    for test_point in X_test:
        distances = [euclidean_distance(train_point, test_point) for train_point in X_train]
        k_indices = np.argsort(distances)[:k]
        k_nearest_targets = y_train[k_indices]
        y_pred.append(np.mean(k_nearest_targets))
    return np.array(y_pred)
In [93]: num_samples = len(df)
train_size = int(0.8 * num_samples)
indices = np.random.permutation(num_samples)
train_indices = indices[:train_size]
test_indices = indices[train_size:]
In [94]: X = df[['X1']]
y = df['Y']
X_train, X_test = X.values[train_indices], X.values[test_indices]
y_train, y_test = y.values[train_indices], y.values[test_indices]
In [95]: k_value = 3
y_pred = knn_regression(X_train, y_train, X_test, k=k_value)
In [96]: mse = np.mean((y_test - y_pred) ** 2)
print(f'Mean Squared Error (MSE) for k={k_value}: {mse}')
Mean Squared Error (MSE) for k=3: 7.283387910025556


KNN Implementation On Given Dataset (11067) Last Checkpoint: 2 hours ago (autosaved)
Mean Squared Error (MSE) for k=3: 7.283387910025556
In [97]: plt.figure(figsize=(8, 6))
plt.scatter(X_train, y_train, color='blue', label='Training Data')
plt.scatter(X_test, y_test, color='green', label='Testing Data')
plt.scatter(X_test, y_pred, color='red', label='Predictions')
plt.xlabel('X1')
plt.ylabel('Y')
plt.title(f'KNN Regression (k={k_value}), MSE: {mse:.2f}')
plt.legend()
plt.show()
KNN Regression (k=3), MSE: 7.28
Training Data
Testing Data
Predictions
Y
X1