Subject: Data Mining and Machine Learning

Project: Mini Project 1

Z Number: Z23737910

Name: Aditya Pradeep Waghmode

Google Colab Link : https://colab.research.google.com/drive/16_iT7Q_KHdBP7-DHJqvbuKQNryd-OG8m?usp=sharing

Answer 1.a:

Importing Numpy for mathematical calculations and importing matplotlib.pyplot for plotting the graph. Also, Defining given data d generated output and x input data.

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

# Given data
d = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7972,
11.6371])
x = np.array([1, 1.7143, 2.4286, 3.1429, 3.8571, 4.5714, 5.4857, 6])
```

Calculating coefficient of linear regression. As we are using the equation y=mx+b calculating m(slope) and b(y-intercept) for calculating these we need x_mean(x_dash) and d_mean(d_dash)

$$\text{Slope (m)} = \frac{\sum (x-\bar{x})(y-\bar{y})}{\sum (x-\bar{x})^2}$$

---

y = mx + b (line equation)

y-intercept(b) = y_mean * (slope * x_mean)

```
n = len(x)
x_dash = np.mean(x)
d_dash = np.mean(d)
m = np.sum((x - x_dash) * (d - d_dash)) / np.sum((x - x_dash)**2)
b = d_dash - m * x_dash
```

```
print("Slope:",m)
print("Y-intercept:",b)
```

```
Slope: 1.2443510770486852
Y-intercept: 6.232149953403384
```

Calculating the MSE(Mean Squared Error) to calculate the mse we need to find the predicted d values. Here we have 'n' number of inputs y which is our array d.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

```
d_pred = b + m*x
mse = np.mean((d - d_pred)**2)
mse = np.round(mse,4)
print("MSE(for simple linear regression):", mse)
```

```
MSE(for simple linear regression): 1.7873
```

Now plotting the line and the data point in a graph using matpoltlib to visualize the simple linear regression.

```
plt.scatter(x , d , label = 'Data Points' , color = 'orange')
plt.plot(x , d_pred , label = 'Linear Regression Line' , color = 'green')
plt.xlabel('x: input data')
plt.ylabel('d: genrated output')
plt.legend()
plt.title(f'Simple Linear Regression (MSE:{mse})')
plt.show()
```

Answer 1.b:

To find the second order polynomial we create a matrix A which will contain ones, x and x^2 as we are dealing with 2nd order polynomial. Matrix A is also called Vandermonde matrix. Then we will be using ordinary least squares estimation (mentioning the equation below) to calculate the coefficients of the quadratic equation, where we will consider the coefficient as 'beta'

$$\vec{\hat{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\,\mathbf{X}^\mathsf{T}\vec{y}$$

```
A = np.column_stack((np.ones(n) , x , x**2))
print("Vandermonde matrix:\n",A)

beta = np.linalg.inv(A.T @ A) @ A.T @ d
print("Coefficients of the quadratic equation:\n", beta)
```

```
Vandermonde matrix:
 [[ 1.          1.          1.        ]
 [ 1.          1.7143      2.93882449]
 [ 1.          2.4286      5.89809796]
 [ 1.          3.1429      9.87782041]
 [ 1.          3.8571     14.87722041]
 [ 1.          4.5714     20.89769796]
 [ 1.          5.4857     30.09290449]
 [ 1.          6.         36.        ]]
Coefficients of the quadratic equation:
 [ 1.09191476  4.98061174 -0.52837073]
```

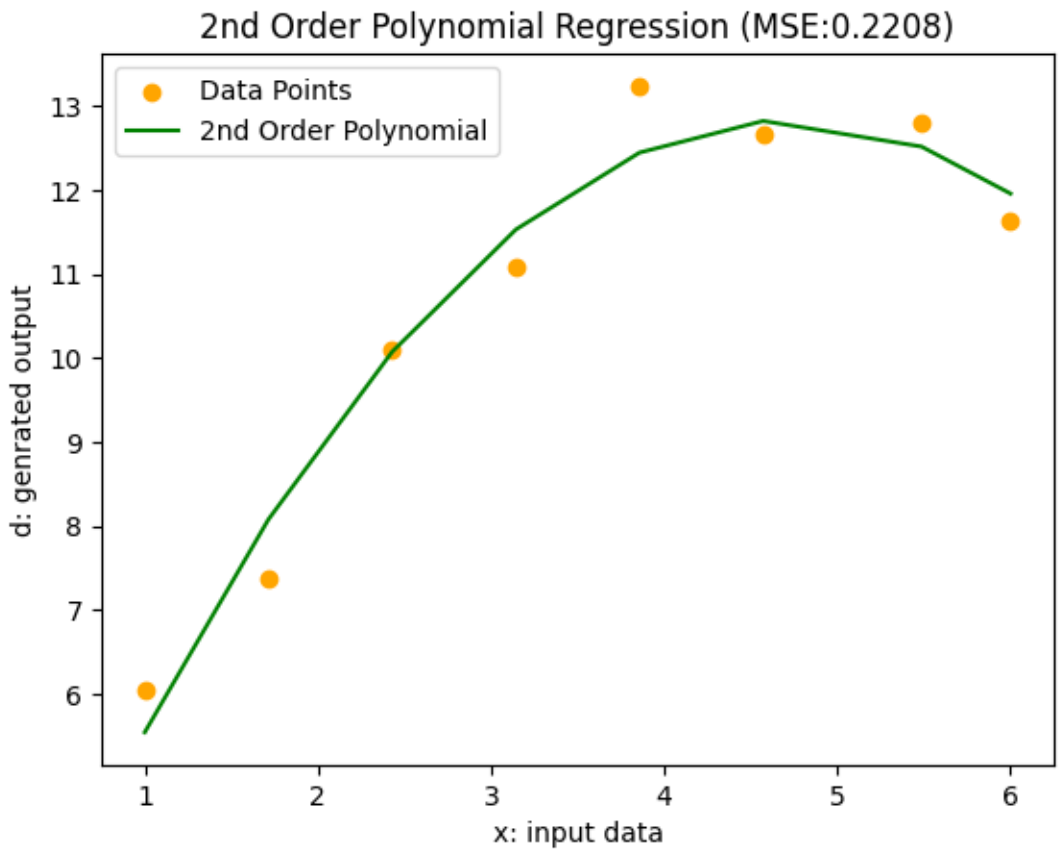
Now, as we have the coefficients, we can calculate the predicted d values and the mean squared error for the quadratic equation.

```
d_pred_poly2 = A @ beta
mse_poly2 = np.mean((d - d_pred_poly2)**2)
mse_poly2 = np.round(mse_poly2, 4)
print("MSE(mean squared error) for 2nd order polynomial: ", mse_poly2)
```

```
MSE(mean squared error) for 2nd order polynomial:  0.2208
```

Now, plotting the line and the data point in a graph to visualize the regression of second order polynomial equation.

```
plt.scatter(x , d , label="Data Points" , color = 'orange')
plt.plot(x ,  d_pred_poly2 , label = "2nd Order Polynomial" , color =
'green')
plt.xlabel('x: input data')
plt.ylabel('d: genrated output')
plt.legend()
plt.title(f'2nd Order Polynomial Regression (MSE:{mse_poly2})')
plt.show()
```



Answer 1.c:

Now, as we solved 2nd order polynomial equation, we will solve the 6th order polynomial equation using the Vandermonde matrix and the ordinary least square estimation to calculate the coefficients and later predicted d values and the mean squared error.

```
A_poly6 = np.column_stack((np.ones(n) , x , x**2 , x**3 , x**4 , x**5 , x**6 ))
print("Vandermonde matrix for 6th order polynomial:\n", A_poly6)
beta_poly6 = np.linalg.inv(A_poly6.T @ A_poly6) @ A_poly6.T @ d
print("Coefficients of 6th order polynomial:\n",beta_poly6)
```

```
Vandermonde matrix for 6th order polynomial:
 [[1.00000000e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
  1.00000000e+00 1.00000000e+00 1.00000000e+00]
 [1.00000000e+00 1.71430000e+00 2.93882449e+00 5.03802682e+00
  8.63668938e+00 1.48058766e+01 2.53817143e+01]
 [1.00000000e+00 2.42860000e+00 5.89809796e+00 1.43241207e+01
  3.47875595e+01 8.44850671e+01 2.05180434e+02]
 [1.00000000e+00 3.14290000e+00 9.87782041e+00 3.10450018e+01
  9.75713361e+01 3.06656952e+02 9.63792135e+02]
 [1.00000000e+00 3.85710000e+00 1.48772204e+01 5.73829268e+01
```

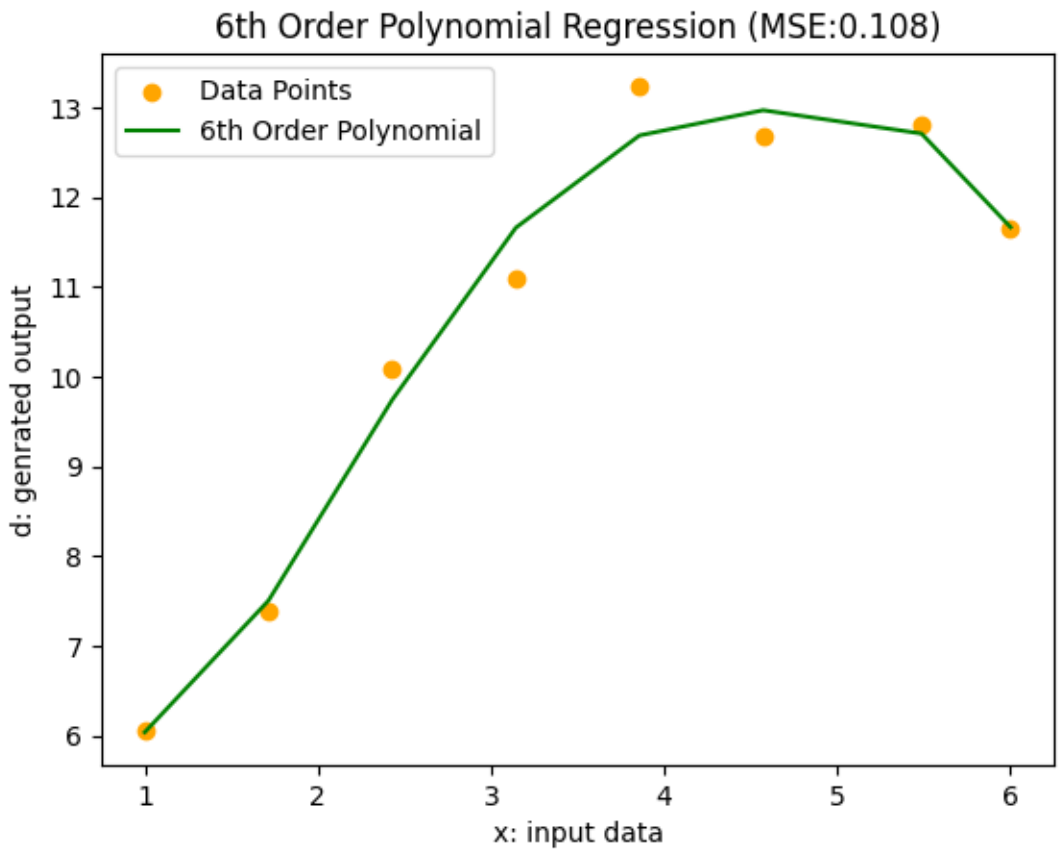
```
  2.21331687e+02 8.53698450e+02 3.29280029e+03]
 [1.00000000e+00 4.57140000e+00 2.08976980e+01 9.55317365e+01
  4.36713780e+02 1.99639337e+03 9.12631267e+03]
 [1.00000000e+00 5.48570000e+00 3.00929045e+01 1.65080646e+02
  9.05582901e+02 4.96775612e+03 2.72516197e+04]
 [1.00000000e+00 6.00000000e+00 3.60000000e+01 2.16000000e+02
  1.29600000e+03 7.77600000e+03 4.66560000e+04]]
Coefficients of 6th order polynomial:
 [ 6.22958478 -0.59818295 -1.05650772  2.20052628 -0.86595403  0.13410632
 -0.00747583]
```

```
d_pred_poly6 = A_poly6 @ beta_poly6
mse_poly6 = np.mean((d - d_pred_poly6)**2)
mse_poly6 = np.round(mse_poly6, 4)
print("MSE for 6th order polynomial:", mse_poly6)
```

```
MSE for 6th order polynomial: 0.108
```

Now plotting the line and the data point in a graph to visualize the regression of sixth order polynomial equation.

```
plt.scatter(x , d , label = "Data Points" , color = 'orange')
plt.plot(x , d_pred_poly6 , label = '6th Order Polynomial' ,
color='green')
plt.xlabel('x: input data')
plt.ylabel('d: genrated output')
plt.legend()
plt.title(f'6th Order Polynomial Regression (MSE:{mse_poly6})')
plt.show()
```



Answer 1.d:

In this problem we are removing the 6 index data point d=12.7972 and x=5.4857 then calculating the 6th order polynomial using the same method as above solution and then we are adding the removed data point again if the mse increases we can say the method we are using is overfitting or else not

```
# Removing the data point
x_remove = np.delete(x , 6)
d_remove = np.delete(d , 6)

# Calculating the Vandermonde matrix when the one data point is removed
A_poly6_remove = np.column_stack((np.ones(n-1) , x_remove , x_remove**2 ,
x_remove**3 , x_remove**4 , x_remove**5 , x_remove**6))
print("Removed data point Vandermonde matrix:\n", A_poly6_remove)

# Calculating the coefficient when the one data point is removed
beta_poly6_remove = np.linalg.inv(A_poly6_remove.T @ A_poly6_remove) @
A_poly6_remove.T @ d_remove
print("Removed data point Coefficient:\n",beta_poly6_remove)
```

```
# Calculating the predicted values of d and mse when the data point is
removed
d_pred_poly6_remove = A_poly6 @ beta_poly6_remove
mse_poly6_remove = np.mean((d_pred_poly6_remove - d)**2)
mse_poly6_remove = np.round(mse_poly6_remove)
print("MSE for 6th polynomial order when one data point is removed: ",
mse_poly6_remove)
```

```
Removed data point Vandermonde matrix:
 [[1.00000000e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
  1.00000000e+00 1.00000000e+00 1.00000000e+00]
 [1.00000000e+00 1.71430000e+00 2.93882449e+00 5.03802682e+00
  8.63668938e+00 1.48058766e+01 2.53817143e+01]
 [1.00000000e+00 2.42860000e+00 5.89809796e+00 1.43241207e+01
  3.47875595e+01 8.44850671e+01 2.05180434e+02]
 [1.00000000e+00 3.14290000e+00 9.87782041e+00 3.10450018e+01
  9.75713361e+01 3.06656952e+02 9.63792135e+02]
 [1.00000000e+00 3.85710000e+00 1.48772204e+01 5.73829268e+01
  2.21331687e+02 8.53698450e+02 3.29280029e+03]
 [1.00000000e+00 4.57140000e+00 2.08976980e+01 9.55317365e+01
  4.36713780e+02 1.99639337e+03 9.12631267e+03]
 [1.00000000e+00 6.00000000e+00 3.60000000e+01 2.16000000e+02
  1.29600000e+03 7.77600000e+03 4.66560000e+04]]
Removed data point Coefficient:
 [ 9.96163570e+01 -2.44784110e+02  2.42504670e+02 -1.17782523e+02
  3.02041627e+01 -3.90495113e+00  1.99586064e-01]
MSE for 6th polynomial order when one data point is removed:  11.0
```

```
# Adding the removed data point back
x_add = np.insert(x_remove, 6, 5.2857)
d_add = np.insert(d_remove, 6, 12.7772)

# Calculating the Vandermonde matrix when the one data point is added back
A_poly6_add = np.column_stack((np.ones(n), x_add, x_add**2, x_add**3,
x_add**4, x_add**5, x_add**6))
print("Added data point Vandermonde matrix:\n", A_poly6_add)

# Here we are using the the 'beta_poly6' as the calculation would be
# the same for the coefficient when the data point is added back

# Calculating the predicted values of d and mse when the data point is
added back
d_pred_poly6_add = A_poly6_add @ beta_poly6
mse_poly6_add = np.mean((d_pred_poly6_add - d_add)**2)
mse_poly6_add = np.round(mse_poly6_add,4)
print("MSE for 6th polynomial order when one data point is added back: ",
mse_poly6_add)
```

```
Added data point Vandermonde matrix:
 [[1.00000000e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
  1.00000000e+00 1.00000000e+00 1.00000000e+00]
```

```
 [1.00000000e+00 1.71430000e+00 2.93882449e+00 5.03802682e+00
  8.63668938e+00 1.48058766e+01 2.53817143e+01]
 [1.00000000e+00 2.42860000e+00 5.89809796e+00 1.43241207e+01
  3.47875595e+01 8.44850671e+01 2.05180434e+02]
 [1.00000000e+00 3.14290000e+00 9.87782041e+00 3.10450018e+01
  9.75713361e+01 3.06656952e+02 9.63792135e+02]
 [1.00000000e+00 3.85710000e+00 1.48772204e+01 5.73829268e+01
  2.21331687e+02 8.53698450e+02 3.29280029e+03]
 [1.00000000e+00 4.57140000e+00 2.08976980e+01 9.55317365e+01
  4.36713780e+02 1.99639337e+03 9.12631267e+03]
 [1.00000000e+00 5.28570000e+00 2.79386245e+01 1.47675187e+02
  7.80566738e+02 4.12584161e+03 2.18079610e+04]
 [1.00000000e+00 6.00000000e+00 3.60000000e+01 2.16000000e+02
  1.29600000e+03 7.77600000e+03 4.66560000e+04]]
```

```
MSE for 6th polynomial order when one data point is added back:  0.1076
```

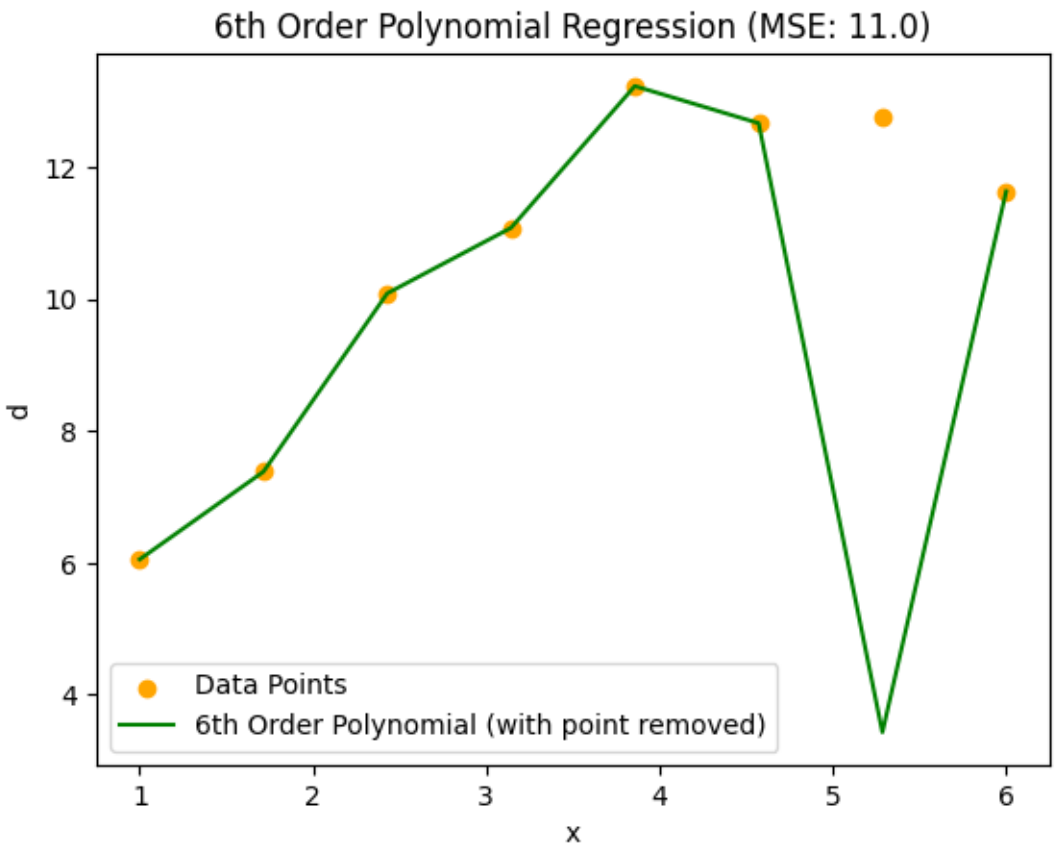
```
# Now to calculate the differnce between the MSE when the data was removed
and added back

change_in_mse = mse_poly6_remove - mse_poly6_add
change_in_mse = np.round(change_in_mse, 4)
print("Change in mse when one data point is removed and added back: ",
change_in_mse)
```

```
Change in mse when one data point is removed and added back:  10.8924
```

We can see the MSE changes when we remove and add the data point which means the method is adjusting the curve perfectly to the given data points which means that the method is overfitting. As result, we can say that the sixth order polynomial tends to overfit the data.

Now we will plot the data for sixth-order polynomial curve with the point added back

```
plt.scatter(x_add, d_add, label='Data Points' , color = 'orange')
plt.plot(x_add, d_pred_poly6_remove, label='6th Order Polynomial (with
point removed)', color='green')
plt.xlabel('x')
plt.ylabel('d')
plt.legend()
plt.title(f'6th Order Polynomial Regression (MSE: {mse_poly6_remove})')
plt.show()
```



As we can see in the graph that for the point we removed the curve is not considering that point which means the model is fitting to the data point given to it while training means we can conclude that the 6th order polynomial equation is overfitting

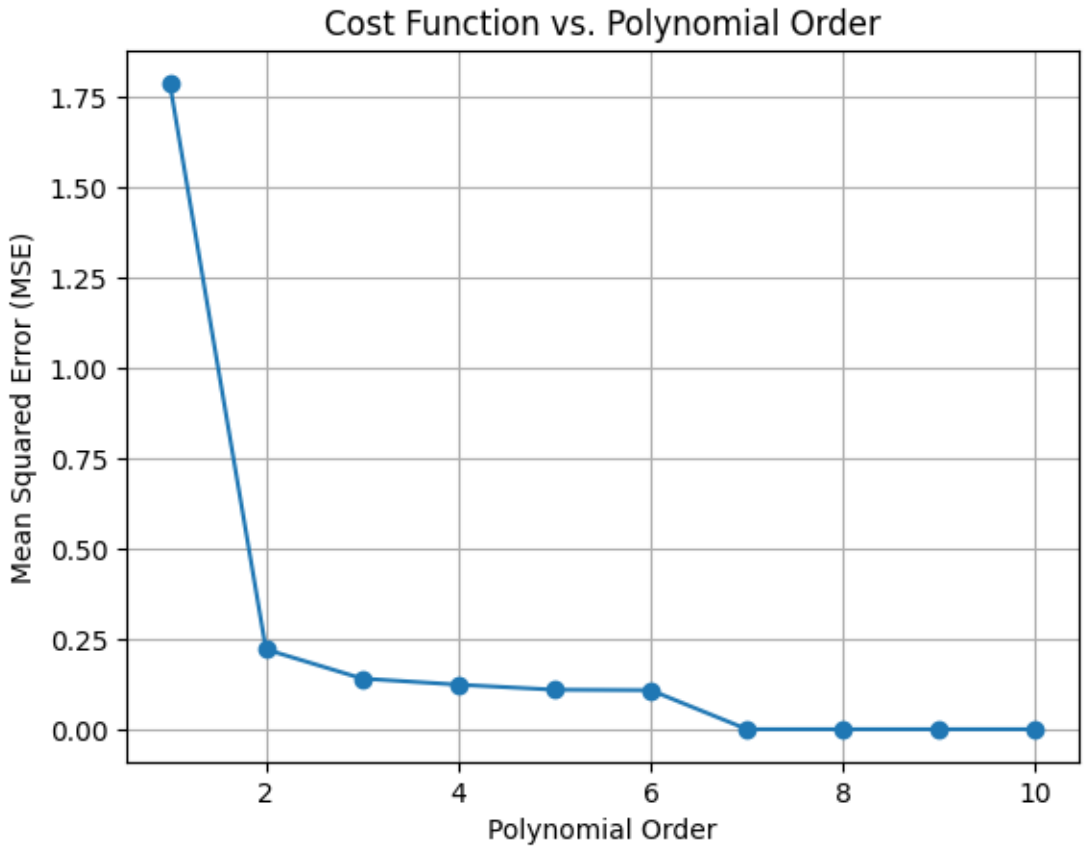
---

Answer 1e:

For calculating the order for 1 to 10 polynomial equations we are using the same methods used in above solutions but here we are using for loop to calculate the Vandermonde matrix, Coefficients, Predicted output values and finally MSE for all the order of polynomial from 1 to 10. Then will plot the mse versus polynomial order graph to find which order will be the better fit.

```
orders = range(1, 11)
mse_values = []

for order in orders:
    # Calculating the Vandermonde matrix
    A_order = np.column_stack([x**i for i in range(order + 1)])  #
Including constant term so adding one to order
```

```
    # Calculating coefficients for the current order using the normal
equation
    beta_order = np.linalg.lstsq(A_order, d, rcond=None)[0]

    # Calculating predicted values
    d_pred_order = A_order @ beta_order

    # Calculate Mean Squared Error (MSE)
    mse_order = np.mean((d_pred_order - d)**2)
    mse_values.append(mse_order)

# Ploting Cost Function (MSE) v/s polynomial order
plt.plot(orders, mse_values, marker='o', linestyle='-')
plt.xlabel('Polynomial Order')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Cost Function vs. Polynomial Order')
plt.grid(True)
plt.show()
```



As we can see the MSE for 1 and 2 order is very high compared to others, for 7 to 10 the MSE is zero which means the order is overfitting. And as we calculated and found that the 6th order

was overfitting as well. The best possible orders would be 3rd, 4th or 5th for the given data but if the data is increased then they might tend to overfit as 6th order polynomial. In that case the 2nd order polynomial might be preferable.

Answer 2a:

In this problem for Logistic regression, we are solving the by defining the Vandermonde matrix with respect to x1 and x2. Initializing weights as 0. Applying Sigmoid function and then calculating gradient descent and then updating the weights for mentioned number of epochs and mentioning the learning rate to learn the descent.

Sigmoid Function (Logistic Function):

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

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

# Given data
x1 = np.array([0.5, 0.8, 0.9, 1.0, 1.1, 2.0, 2.2, 2.5, 2.8, 3.0])
x2 = np.array([0.5, 0.2, 0.9, 0.8, 0.3, 2.5, 3.5, 1.8, 2.1, 3.2])
d = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0])
# Creating Vandermonde matrix
A = np.column_stack((np.ones(len(x1)), x1, x2))
print("Vandermonde matrix:\n",A)
```

```
Vandermonde matrix:
 [[1.  0.5 0.5]
 [1.  0.8 0.2]
 [1.  0.9 0.9]
 [1.  1.  0.8]
 [1.  1.1 0.3]
 [1.  2.  2.5]
 [1.  2.2 3.5]
 [1.  2.5 1.8]
 [1.  2.8 2.1]
 [1.  3.  3.2]]
```

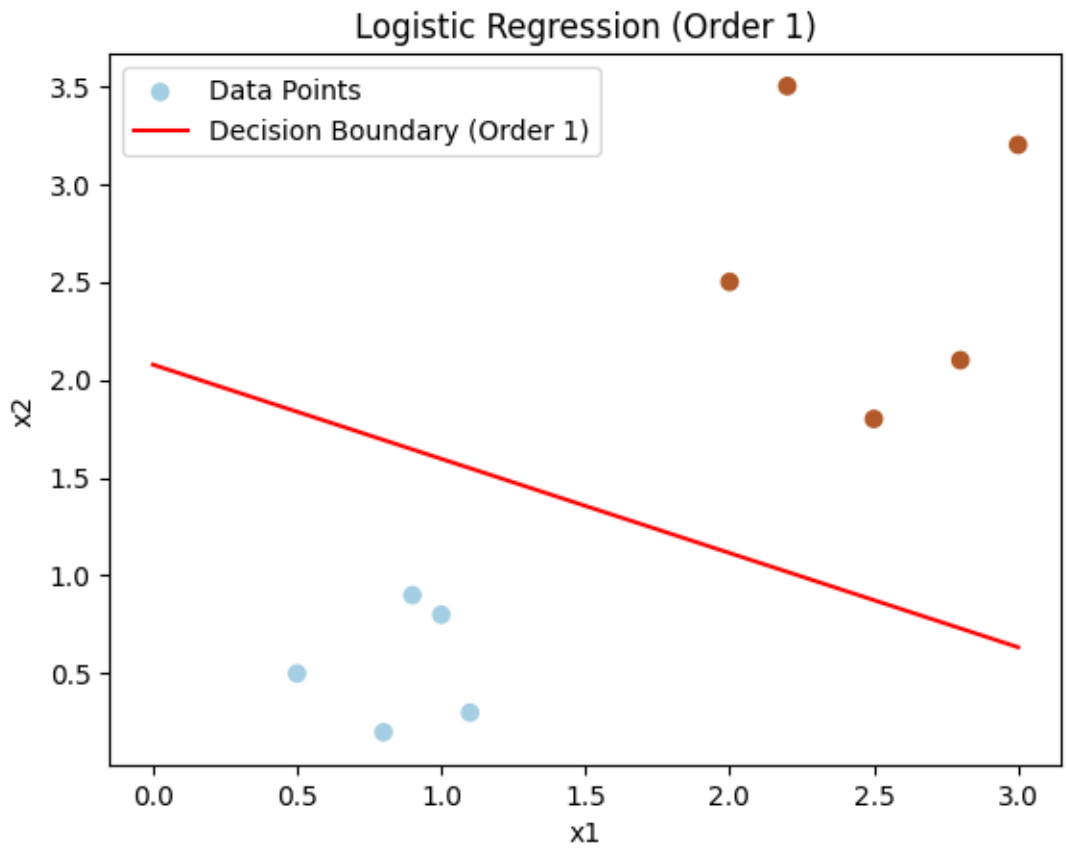
```
# Initializing weights
w = np.zeros(3)
print("Weights:\n",w)
```

```
Weights:
 [0. 0. 0.]
```

```
# Logistic regression function (sigmoidal function)
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
# Applying Gradient descent
lr = 0.01  # Learning rate
epooch = 10000

for _ in range(epooch):
    z = A @ w
    h = sigmoid(z)
    gradient = A.T @ (h - d) / len(d)
    #Gradient descent = (Vandermonde matrix_Transope(Predicted value - Desired value))/length of desired value
    w -= lr * gradient

print("Final Weights:\n",w)
```

```
Final Weights:
 [-5.16911135  1.19713723  2.48809653]
```

```
# Ploting data points and decision boundary(Threshold)
plt.scatter(x1, x2, c=d, cmap=plt.cm.Paired, label='Data Points')
x_boundary = np.linspace(0, 3, 100)
y_boundary = (-w[0] - w[1] * x_boundary) / w[2]
plt.plot(x_boundary, y_boundary, color='red', label='Decision Boundary (Order 1)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Logistic Regression (Order 1)')
plt.show()
```



Answer 2.b:

Calculating the logistic regression by increasing the decision boundary by 2 which will give us the curve while differentiating the class this time. Using the above method as same while updating the Vandermonde matrix by 2.

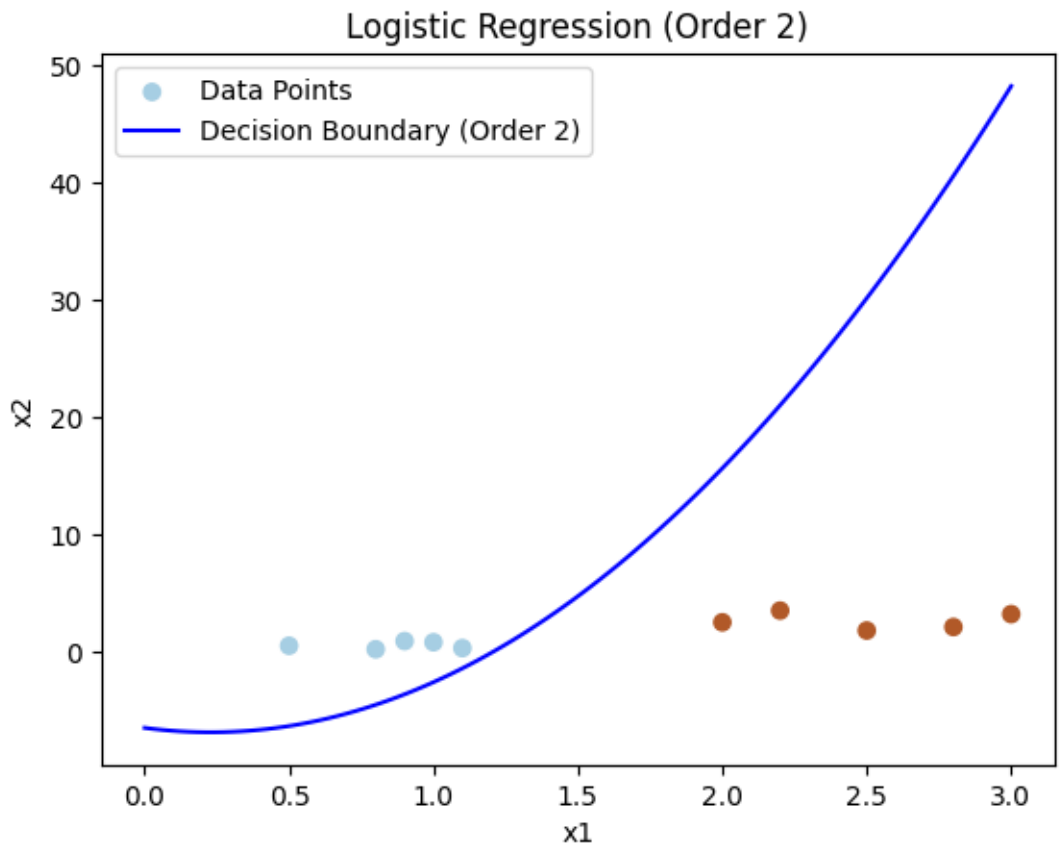
```
# Increasing the order of the decision boundary by 2

# Creating Vandermonde matrix
A_poly2 = np.column_stack((A, x1**2, x2**2, x1 * x2))
print("Vandermonde matrix:\n",A_poly2)
```

```
Vandermonde matrix:
 [[ 1.    0.5   0.5   0.25  0.25  0.25]
 [ 1.    0.8   0.2   0.64  0.04  0.16]
 [ 1.    0.9   0.9   0.81  0.81  0.81]
 [ 1.    1.    0.8   1.    0.64  0.8 ]
 [ 1.    1.1   0.3   1.21  0.09  0.33]
 [ 1.    2.    2.5   4.    6.25  5.  ]
 [ 1.    2.2   3.5   4.84 12.25  7.7 ]
 [ 1.    2.5   1.8   6.25  3.24  4.5 ]
 [ 1.    2.8   2.1   7.84  4.41  5.88]
 [ 1.    3.    3.2   9.   10.24  9.6 ]]
```

```
lr2 = 0.001  # Learning rate
epooch2 = 1500

# Initialize weights
w_poly2 = np.zeros(6)
print("Weights:\n",w_poly2)

for _ in range(epooch2):
    z_poly2 = A_poly2 @ w_poly2
    h_poly2 = sigmoid(z_poly2)
    gradient_poly2 = A_poly2.T @ (h_poly2 - d) / len(d)
    w_poly2 -= lr2 * gradient_poly2

print("Final Weights:\n",w_poly2)
```

```
Weights:
 [0. 0. 0. 0. 0. 0.]
Final Weights:
 [-0.30753645 -0.15484758 -0.04725186  0.12991768  0.24758351  0.20930213]
```

```
# Plot data points and decision boundary for the second-order boundary
plt.scatter(x1, x2, c=d, cmap=plt.cm.Paired, label='Data Points')
x_boundary_poly2 = np.linspace(0, 3.0, 100)
y_boundary_poly2 = (-w_poly2[0] - w_poly2[1] * x_boundary_poly2 -
w_poly2[3] * x_boundary_poly2**2 - w_poly2[5] * x_boundary_poly2**2) /
w_poly2[2]
plt.plot(x_boundary_poly2, y_boundary_poly2, color='blue', label='Decision
Boundary (Order 2)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Logistic Regression (Order 2)')
plt.show()
```



Logistic Regression (Order 2) boundary indicates a more complicated model that can capture non-linear relationships, whereas a Logistic Regression (Order 1) boundary represents a simple model that assumes a linear relationship between variables. The non-linear model highlights the trade-offs between model simplicity, fit, and generalizability. The linear model fits the given data better at the expense of an increased risk of overfitting and decreased interpretability. The linear model is easier to interpret and requires less computing power, but it may underfit complex data patterns.