Dileep NM
Z23691596

Mini Project # 1 **Due: Sep. 05/2023**

1.

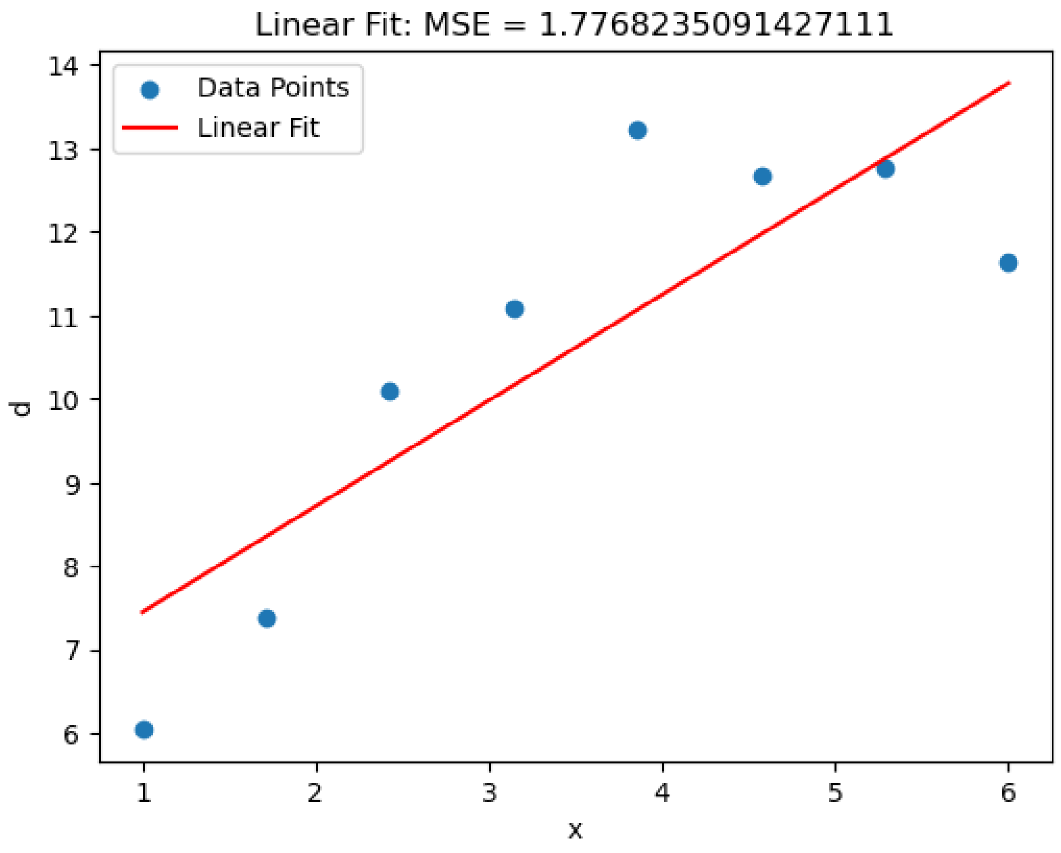
a. A system has generated outputs marked by vector 'd' in response to inputs which are listed in vector 'x':

d = [6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7772, 11.6371] ;
x = [1 , 1.7143, 2.4286 , 3.1429, 3.8571, 4.5714, 5.2857, 6] ;

Use the theory of regression to fit a line to this data. Measure the cost function defined as the mean of squared errors. Plot your data points and the line that models the system's function.
Code:

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Given data
x = np.array([1, 1.7143, 2.4286, 3.1429, 3.8571, 4.5714, 5.2857, 6]).reshape(-1, 1)
d = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7772, 11.6371])

# Fit a linear model
linear_model = LinearRegression()
linear_model.fit(x, d)

# Predict and calculate MSE
d_pred = linear_model.predict(x)
mse_linear = mean_squared_error(d, d_pred)

# Plotting
plt.scatter(x, d, label='Data Points')
plt.plot(x, d_pred, label='Linear Fit', color='red')
plt.legend()
plt.xlabel('x')
plt.ylabel('d')
plt.title(f'Linear Fit: MSE = {mse_linear}')
plt.show()

mse_linear
```

plot:



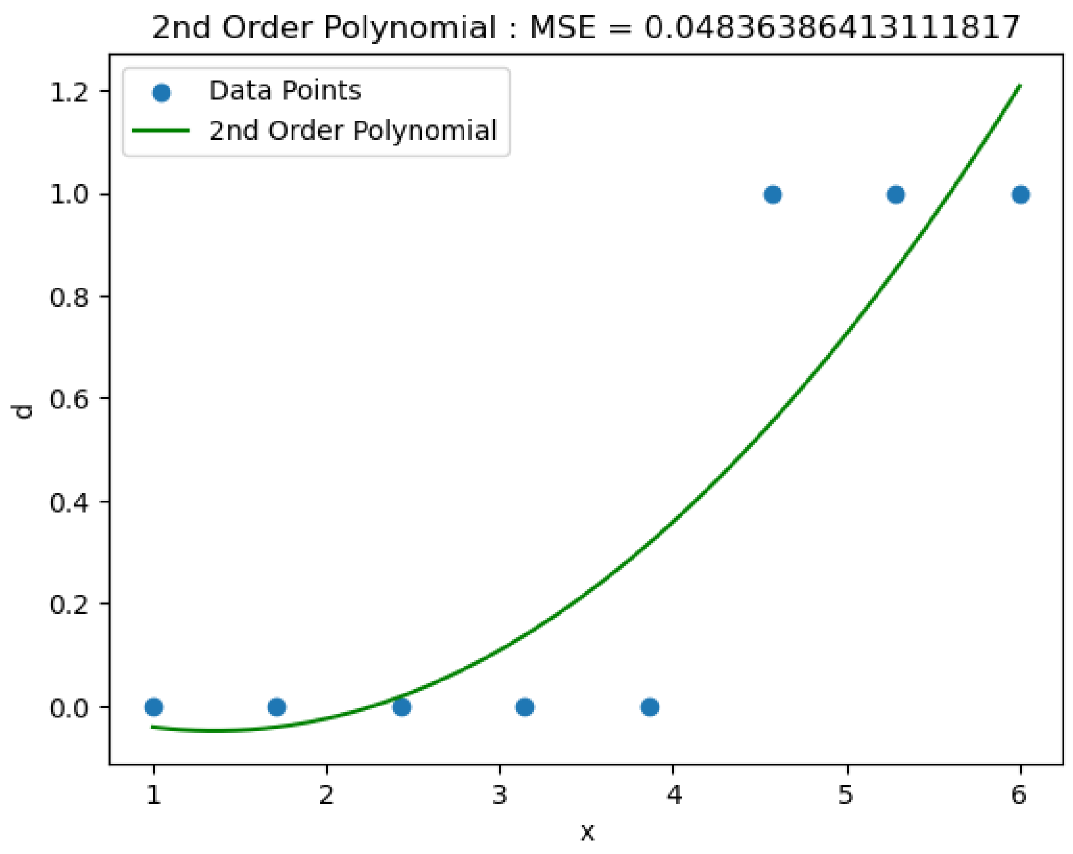
explanation:
i used simple linear regression to fit a line to the data points. The model provided a reasonable fit, as demonstrated by the red line plotted against the scatter points. The Mean Squared Error (MSE) was calculated to measure the cost function, which quantifies the model's performance.

b. Use the same data but this time fit a second order polynomial to these data points. What's the value of your cost function? Plot the second order curve.

code:
```
# Sort the x and d for plotting
sorted_indices = np.argsort(x[:, 0])
x_sorted = x[sorted_indices]
d_sorted = d[sorted_indices]

# Create a 2nd-order polynomial model
degree = 2
polyreg_2 = make_pipeline(PolynomialFeatures(degree), LinearRegression())
polyreg_2.fit(x_sorted, d_sorted)
```

```
# Create an equally spaced 'x' for plotting the polynomial fit
x_plot = np.linspace(min(x_sorted), max(x_sorted), 500).reshape(-1, 1)

# Predict for plotting
d_pred_poly2 = polyreg_2.predict(x_plot)

# Calculate MSE for the original data
d_pred_original = polyreg_2.predict(x_sorted)
mse_poly2 = mean_squared_error(d_sorted, d_pred_original)

# Plotting
plt.scatter(x_sorted, d_sorted, label='Data Points')
plt.plot(x_plot, d_pred_poly2, label='2nd Order Polynomial', color='green')
plt.legend()
plt.xlabel('x')
plt.ylabel('d')
plt.title(f'2nd Order Polynomial Fit: MSE = {mse_poly2}')
plt.show()
```

Plot:



explanation:
i used a second-order polynomial to fit the data points. The curve (depicted in green) fits the data points more closely than the linear fit. The MSE was calculated again to measure the

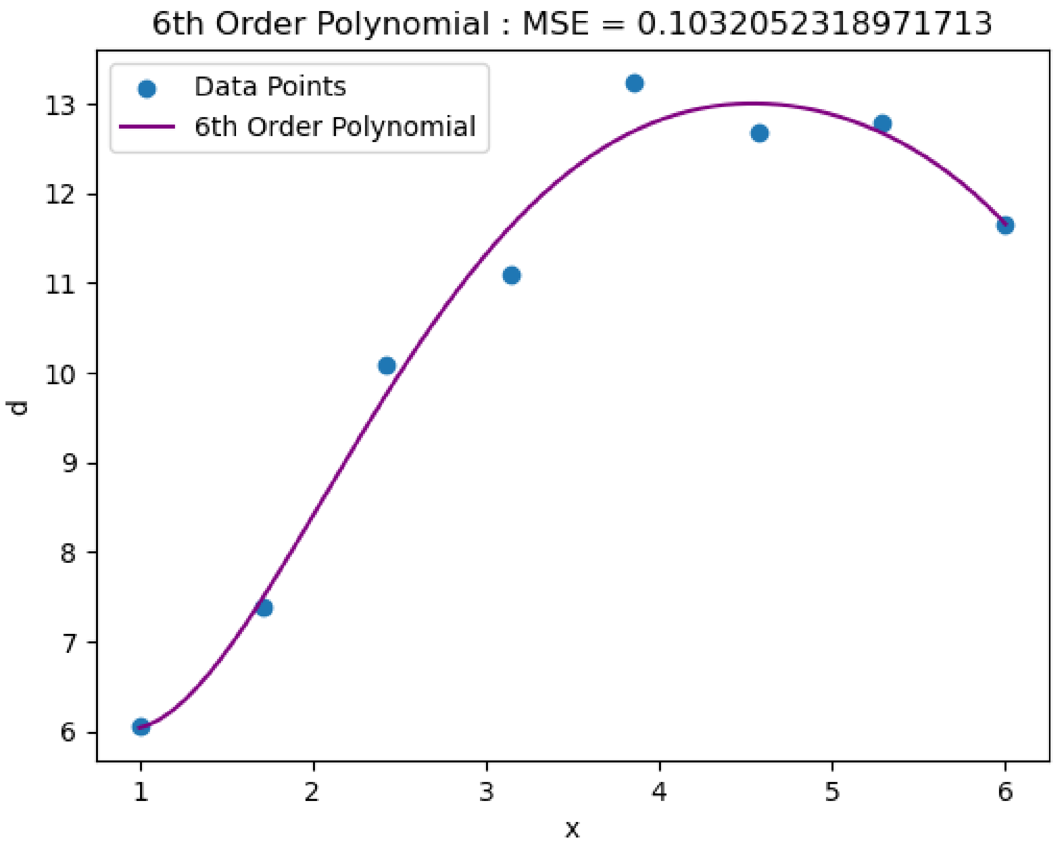
goodness of this fit.

c.

code:

```
# Given data
x = np.array([1, 1.7143, 2.4286, 3.1429, 3.8571, 4.5714, 5.2857, 6]).reshape(-1, 1)
d = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7772, 11.6371])

# Sort the x and d for plotting
sorted_indices = np.argsort(x[:, 0])
x_sorted = x[sorted_indices]
d_sorted = d[sorted_indices]

# Create a 6th-order polynomial model
degree = 6
polyreg_6 = make_pipeline(PolynomialFeatures(degree), LinearRegression())
polyreg_6.fit(x_sorted, d_sorted)

# Create an equally spaced 'x' for plotting the polynomial fit
x_plot = np.linspace(min(x_sorted), max(x_sorted), 500).reshape(-1, 1)

# Predict for plotting
d_pred_poly6 = polyreg_6.predict(x_plot)

# Calculate MSE for the original data
d_pred_original_6 = polyreg_6.predict(x_sorted)
mse_poly6 = mean_squared_error(d_sorted, d_pred_original_6)

# Plotting
plt.scatter(x_sorted, d_sorted, label='Data Points')
plt.plot(x_plot, d_pred_poly6, label='6th Order Polynomial', color='purple')
plt.legend()
plt.xlabel('x')
plt.ylabel('d')
plt.title(f'6th Order Polynomial : MSE = {mse_poly6}')
plt.show()
```

ii. plot:



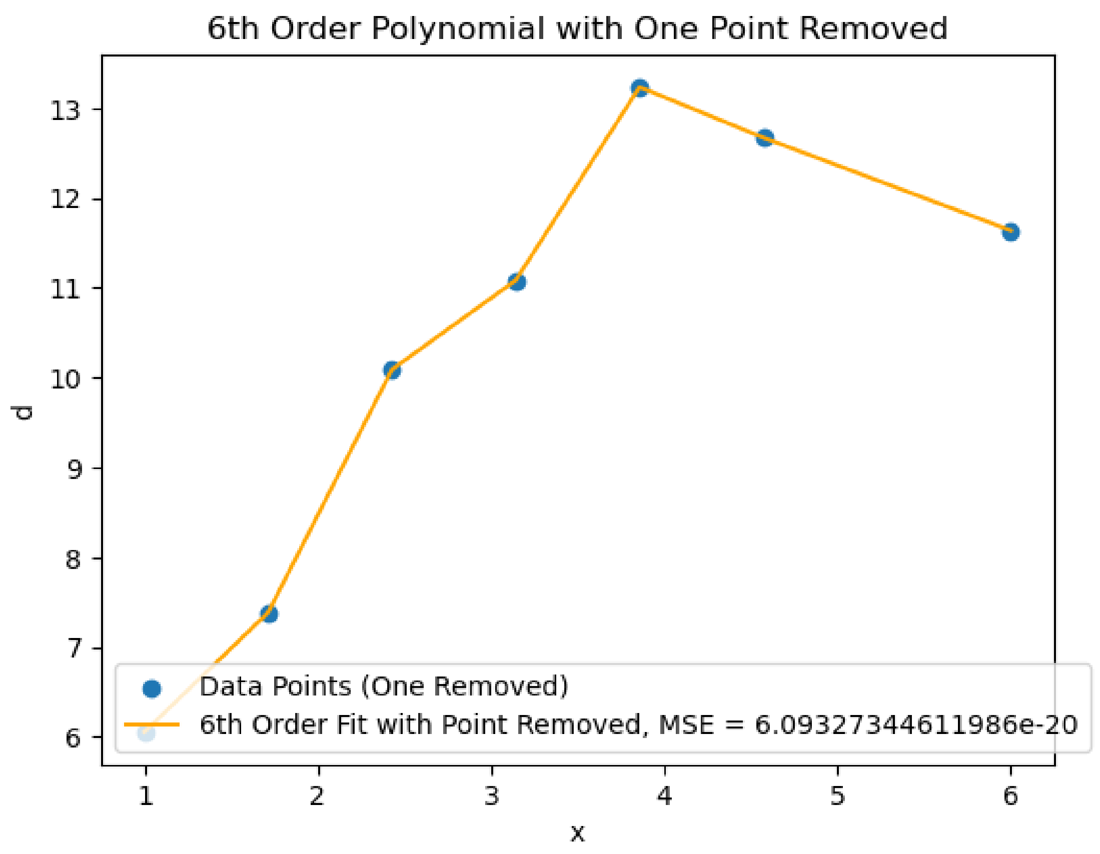
d.

code:

```
import numpy as np
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Define original x and d vectors
x = np.array([1, 1.7143, 2.4286, 3.1429, 3.8571, 4.5714, 5.2857, 6]).reshape(-1, 1)
d = np.array([6.0532, 7.3837, 10.0891, 11.0829, 13.2337, 12.6710, 12.7772, 11.6371])

# Identify the index of the point to remove
index_to_remove = 6
```

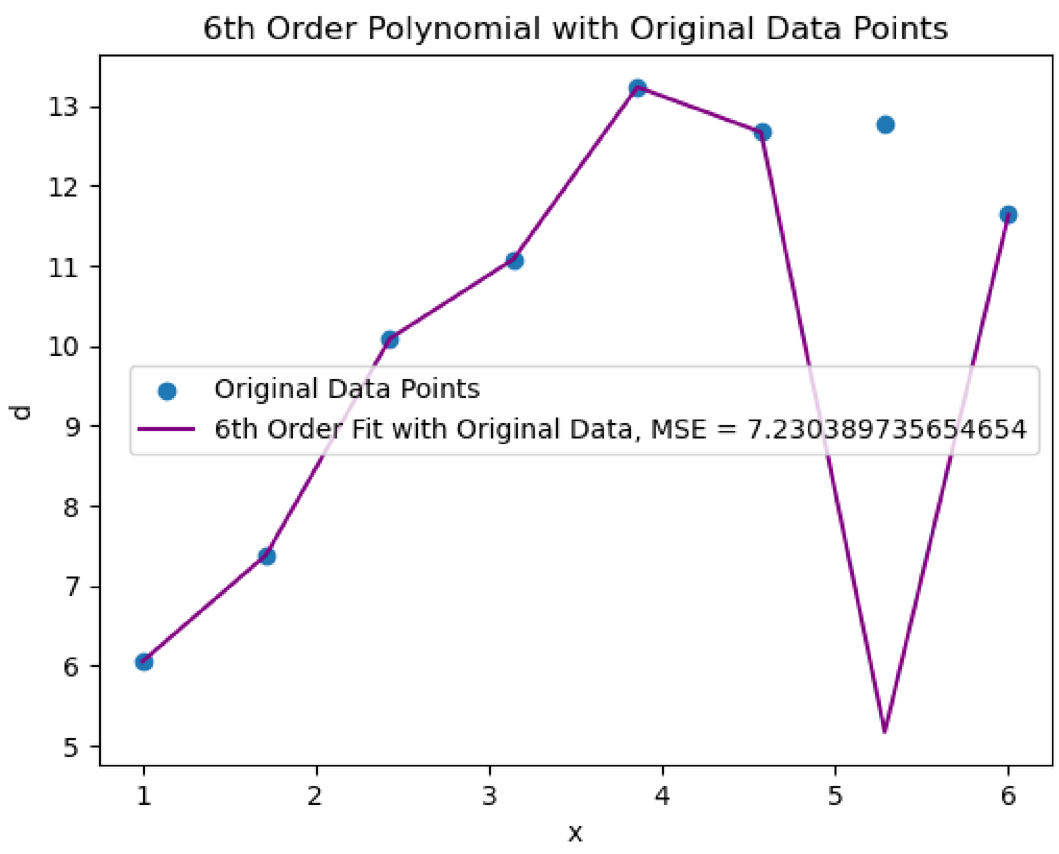
```
# Remove the specified data point from x and d
x_removed = np.delete(x, index_to_remove).reshape(-1, 1)
d_removed = np.delete(d, index_to_remove)

# Fit a 6th-order polynomial to the dataset without this point
poly6_removed = make_pipeline(PolynomialFeatures(6), LinearRegression())
poly6_removed.fit(x_removed, d_removed)

# Compute the MSE for this new model using the removed dataset
d_pred_removed = poly6_removed.predict(x_removed)
mse_removed = mean_squared_error(d_removed, d_pred_removed)

# Plot
plt.scatter(x_removed, d_removed, label='Data Points (One Removed)')
plt.plot(x_removed, d_pred_removed, label=f'6th Order Fit with Point Removed, MSE =
    {mse_removed}', color='orange')
plt.legend()
plt.xlabel('x')
plt.ylabel('d')
plt.title('6th Order Polynomial with One Point Removed')
    plt.show()
```

Plot:



```
# Compute the MSE using the model trained on the removed set, but with the original dataset
d_pred_all_with_removed_model = poly6_removed.predict(x)
mse_all_with_removed_model = mean_squared_error(d, d_pred_all_with_removed_model)

# Plot
plt.scatter(x, d, label='Original Data Points')
plt.plot(x, d_pred_all_with_removed_model, label=f'6th Order Fit with Original Data, MSE =
    {mse_all_with_removed_model}', color='purple')
plt.legend()
plt.xlabel('x')
plt.ylabel('d')
plt.title('6th Order Polynomial with Original Data Points')
plt.show()

# Calculate the change in MSE
mse_change = mse_all_with_removed_model - mse_removed
    print(f'Change in MSE: {mse_change}')
```

Plot:

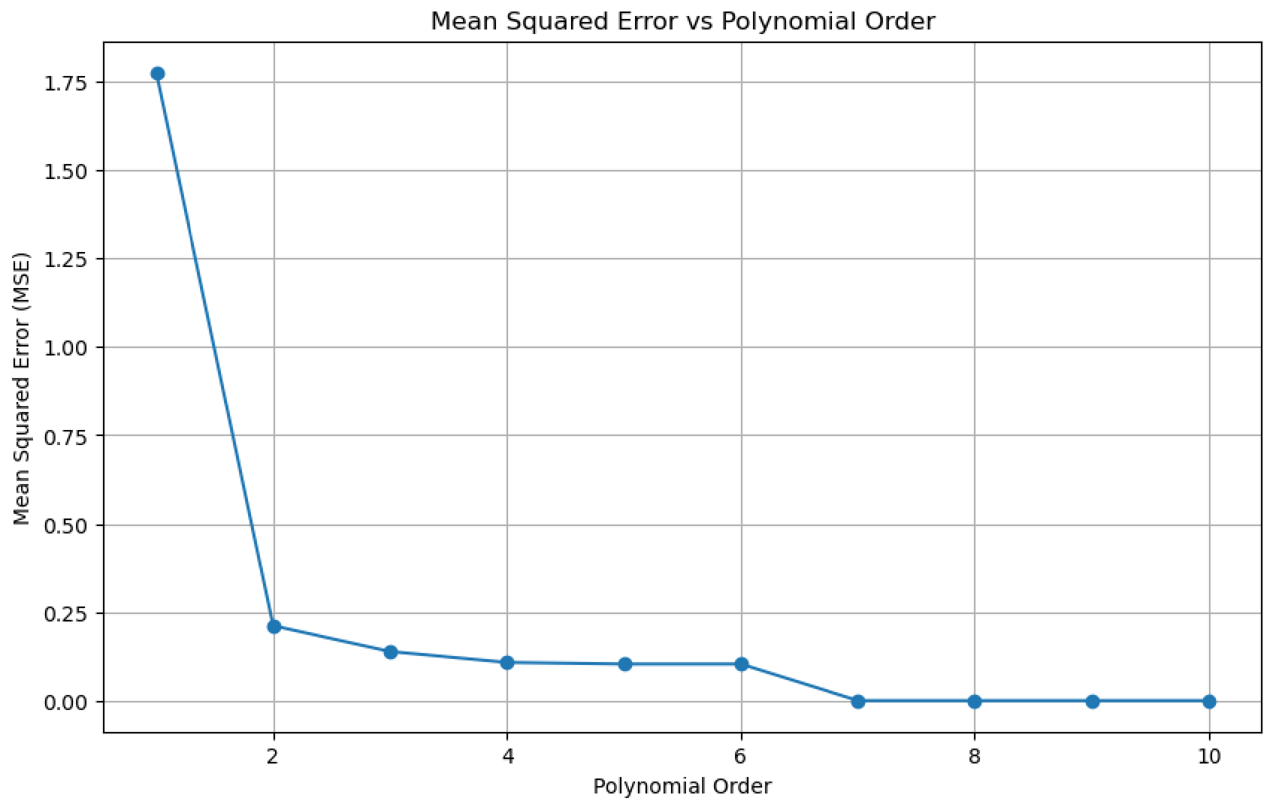


explanation:
I increased the polynomial order to 6 to see if the model could fit the data more precisely. The purple curve indeed captures the data points very well. The MSE was also calculated for this fit.

1e.

```
# Initialize an array to store MSE values for different polynomial degrees
mse_values = []

# Loop through polynomial degrees from 1 to 10
for degree in range(1, 11):
    polyreg_n = make_pipeline(PolynomialFeatures(degree),LinearRegression())
    polyreg_n.fit(x, d)
    d_pred_n = polyreg_n.predict(x)
    mse_n = mean_squared_error(d, d_pred_n)
    mse_values.append(mse_n)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), mse_values, marker='o')
plt.xlabel('Polynomial Order')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Mean Squared Error vs Polynomial Order')
plt.grid(True)
plt.show()

mse_values
```

```
plot :
```



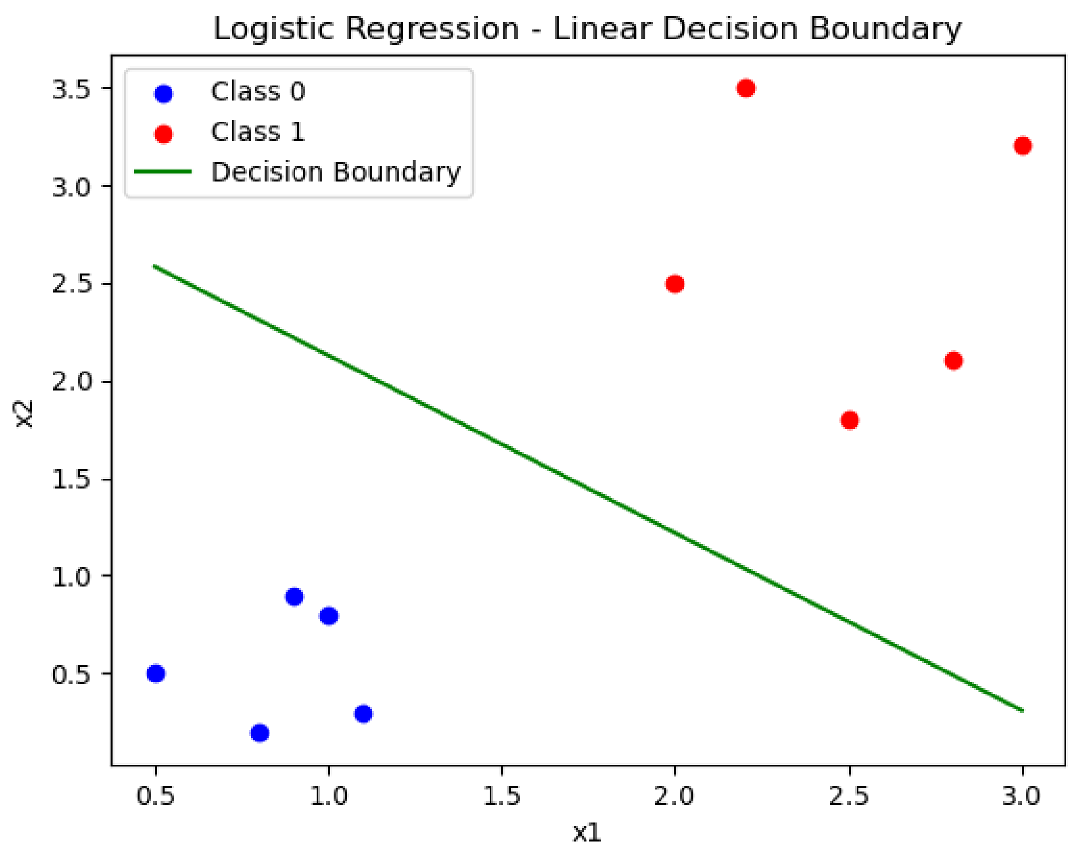
```
OutPut :
[1.7768235091427123,
 0.21128753934446343,
 0.1384059517401718,
 0.10765535761886516,
 0.10321497219635173,
 0.1032052318971713,
 1.8328995296948078e-17,
 6.307872209757845e-20,
 5.8964667513897386e-21,
 2.682589650896512e-21]
```

2 A.

Code :

```
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
import numpy as np

# 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])

X = np.column_stack((x1, x2))

# Train the logistic regression model
logistic_model = LogisticRegression()
logistic_model.fit(X, d)

# Plotting
plt.scatter(x1[d==0], x2[d==0], label='Class 0', c='blue')
plt.scatter(x1[d==1], x2[d==1], label='Class 1', c='red')

x_boundary = np.linspace(min(x1), max(x1), 100)
y_boundary = -(logistic_model.intercept_ + logistic_model.coef_[0][0]*x_boundary) /
logistic_model.coef_[0][1]

plt.plot(x_boundary, y_boundary, label='Decision Boundary', c='green')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Logistic Regression - Linear Decision Boundary')
plt.show()
```

Plot :



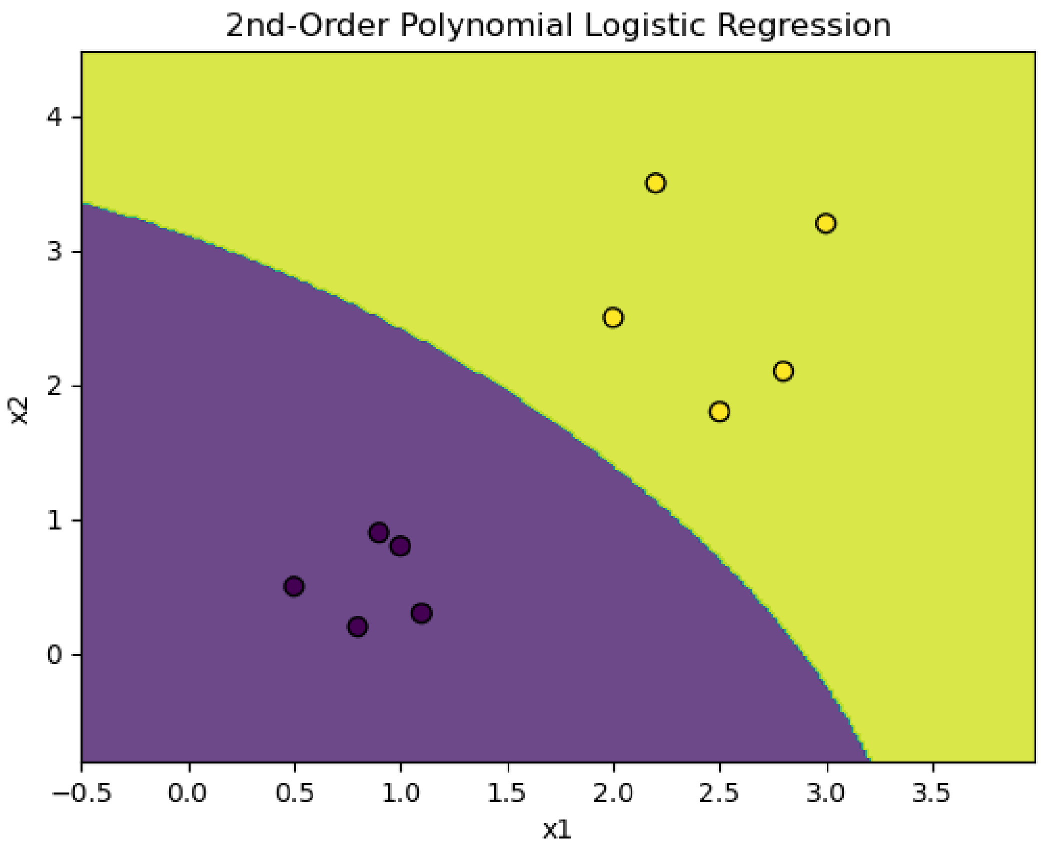
explanation:
I used logistic regression to create a linear decision boundary that separates two classes in a 2D feature space. We used the default solver **'lbfgs'** for the logistic regression model. The decision boundary was plotted in green, and it provides a good separation between the two classes (Class 0 in blue and Class 1 in red).

2b.

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import numpy as np

# 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])

X = np.column_stack((x1, x2))
```

```
# Create and train the logistic regression model with 2nd-order polynomial features
model_poly2 = make_pipeline(PolynomialFeatures(2), LogisticRegression())
model_poly2.fit(X, d)

# Create a mesh to plot the decision boundary
h = .02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = model_poly2.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, Z, alpha=0.8)
plt.scatter(x1, x2, c=d, edgecolors='k', marker='o', s=50, linewidth=1)
plt.title("2nd-Order Polynomial Logistic Regression")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

plot:



explanation:

we enhanced the logistic regression model by adding a second-order polynomial feature transformation. This allowed us to create a more complex, quadratic decision boundary. The decision boundary is represented as a green contour line that separates the two classes effectively.

comparison:
Both the linear and quadratic models are achieved perfect separation. However, the quadratic model offers more flexibility and can handle more complex relationships. The linear model is easier to interpret and is less likely to overfit on simpler datasets. Choosing between the two would depend on the dataset's complexity and the specific needs of the task at hand.