BIRZEIT UNIVERSITY

Electrical and Computer Engineering Department

Machine Learning and Data Science - ENCS5341

Assignment #2

Prepared by: Dana Bornata.

ID: 1200284

Instructor: Dr. Yazan Abu Farha

Date :20/12/2023

Section #1

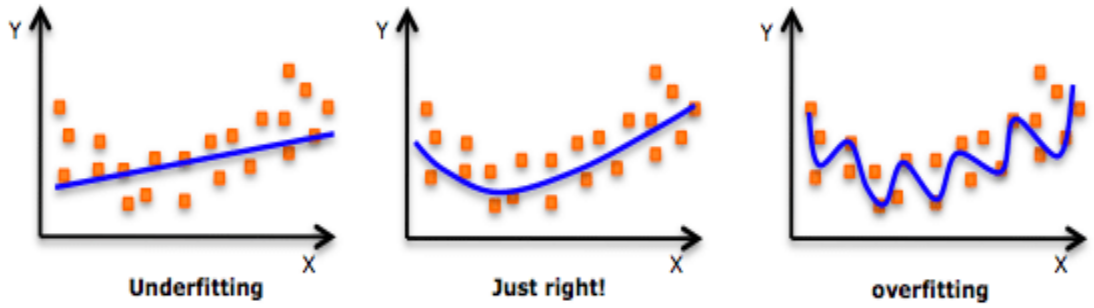


Table of Contents

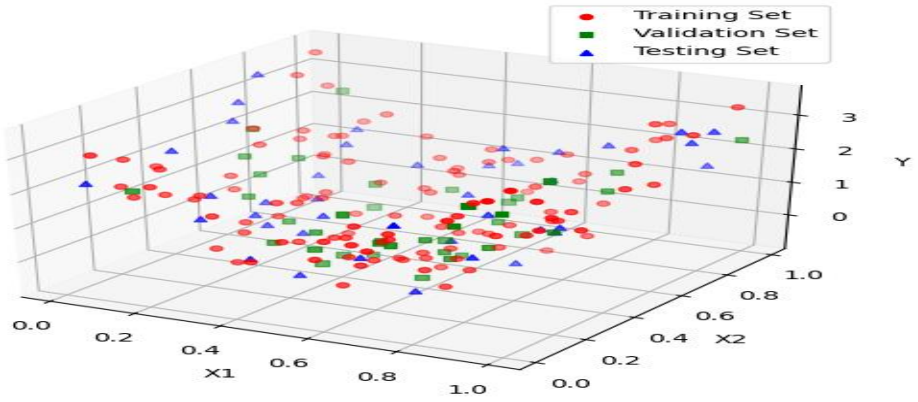
1)Model Selection and Hyper-parameters Tunning

Q1)1

Code:

```
# Question 1.1 Dana Bornata 1200284
data = pd.read_csv('data_reg.csv')
trainingset = data[:120]
validationset = data[120:160]
testing_set = data[160:]

fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(trainingset['x1'], trainingset['x2'], trainingset['y'], c='r', marker='o', label='Training Set')
ax.scatter(validationset['x1'], validationset['x2'], validationset['y'], c='g', marker='s', label='Validation Set')
ax.scatter(testing_set['x1'], testing_set['x2'], testing_set['y'], c='b', marker='^', label='Testing Set')

ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')
ax.legend()
plt.show()
```

Result:



Clarification:

- First, I read the file data, which consists of 3 properties, each property containing 200 data.

- I divided the data into a training set (the first 120 examples), a validation set (the next 40 examples), and a test set (the last 40 examples).

- After that, I represented the data with a 3D diagram, each group with a different color and shape for distinction.

- The X-axis and Y-axis represent the feature X1 and X2, The Z-axis represent the feature y.

Q1)2

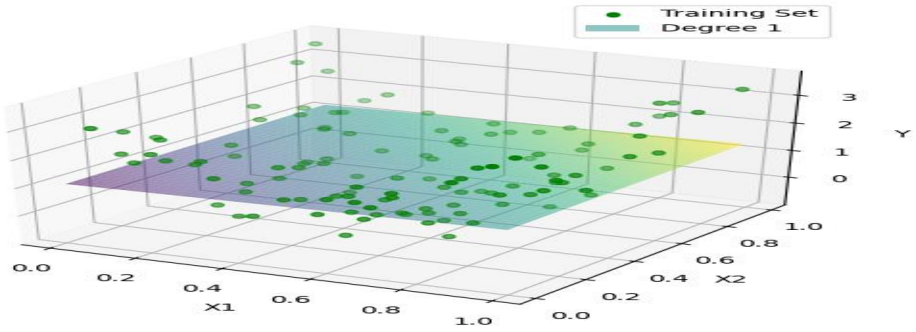
Code:

```
# Question 1.2 Dana Bornata 1200284

Xtrainingset = trainingset[['x1', 'x2']].values
Ytrainingset = trainingset['y'].values
Xvalidationset = validationset[['x1', 'x2']].values
Yvalidationset = validationset['y'].values
COUNT_Degree = np.arange(1, 11)
trainingset_Errors = []
validationset_Errors = []

for degree in COUNT_Degree:
    polynomial = PolynomialFeatures(degree=degree)
    Xtrainingset_polynomial = polynomial.fit_transform(Xtrainingset)
    validationset_polynomial = polynomial.transform(Xvalidationset)
    model = LinearRegression()
    model.fit(Xtrainingset_polynomial, Ytrainingset)
    Ytraining_Predict = model.predict(Xtrainingset_polynomial)
    Yvalidation_Predict = model.predict(validationset_polynomial)
    trainingError = mean_squared_error(Ytrainingset, Ytraining_Predict)
    validationError = mean_squared_error(Yvalidationset, Yvalidation_Predict)
    trainingset_Errors.append(trainingError)
    validationset_Errors.append(validationError)
    Result_figure = plt.figure(figsize=(6, 6))
    ax_surface = Result_figure.add_subplot(111, projection='3d')
    ax_surface.scatter(Xtrainingset[:, 0], Xtrainingset[:, 1], Ytrainingset, c='g', marker='o', label='Training Set')
```

```
    validationset_Errors.append(validationError)
    Result_figure = plt.figure(figsize=(6, 6))
    ax_surface = Result_figure.add_subplot(111, projection='3d')
    ax_surface.scatter(Xtrainingset[:, 0], Xtrainingset[:, 1], Ytrainingset, c='g', marker='o', label='Training Set')
    x1Range = np.linspace(min(Xtrainingset[:, 0]), max(Xtrainingset[:, 0]), 100)
    x2Range = np.linspace(min(Xtrainingset[:, 1]), max(Xtrainingset[:, 1]), 100)
    x1Mesh, x2_mesh = np.meshgrid(x1Range, x2Range)
    X_mesh = np.c_[x1Mesh.ravel(), x2_mesh.ravel()]
    X_mesh_poly = polynomial.transform(X_mesh)
    y_mesh_pred = model.predict(X_mesh_poly)
    ax_surface.plot_surface(x1Mesh, x2_mesh, y_mesh_pred.reshape(x1Mesh.shape), alpha=0.5, cmap='viridis', label=f'Degree {degr
    ax_surface.set_xlabel('X1')
    ax_surface.set_ylabel('X2')
    ax_surface.set_zlabel('Y')
    ax_surface.legend()

    plt.show()

# Plot the validation error vs polynomial degree curve
plt.plot(COUNT_Degree, validationset_Errors, marker='o')
plt.title('validation error vs polynomial degree')
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error')
plt.show()
```

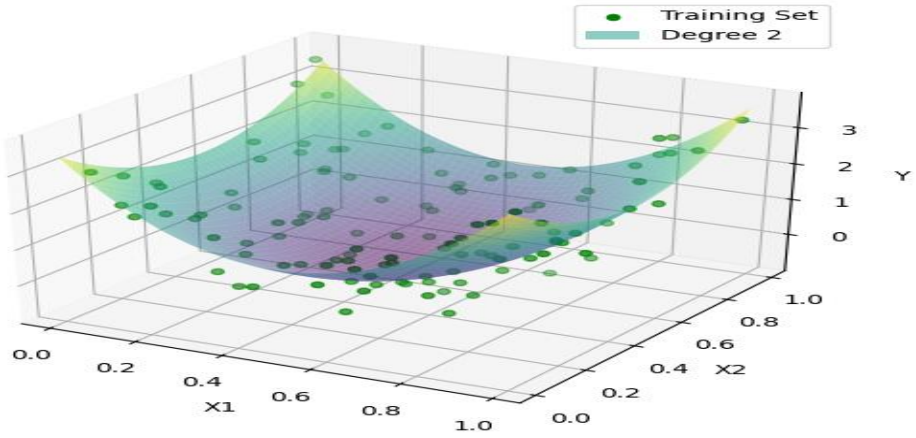
Result and Clarification:

When degree=1 ***Underfitting***



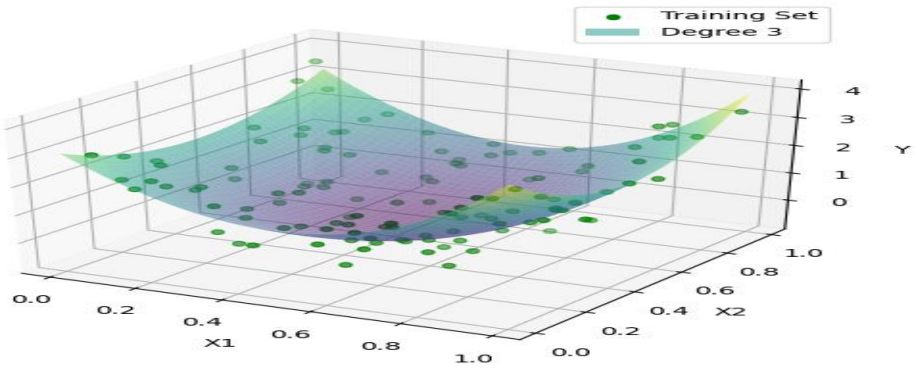
Underfitting is indicated by the first-order polynomial's high MSE of ~ 0.8. A linear model fits the data poorly because it is too basic to identify the underlying patterns in the data. The model isn't complex enough to capture the real relationship.

When degree=2 Best Fit



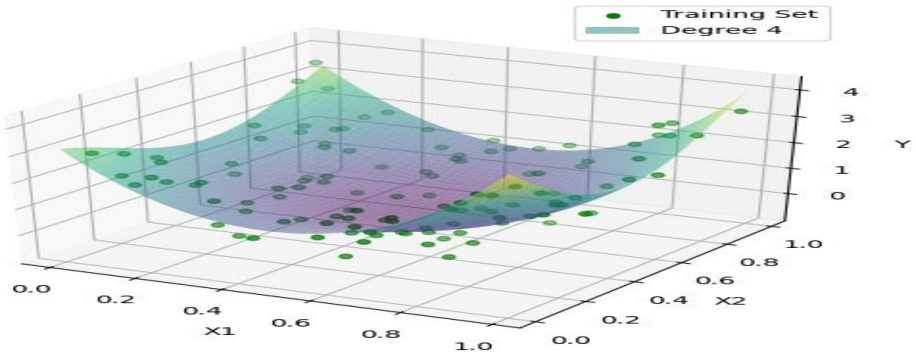
The low MSE of 0.2 for the second-order polynomial indicates a strong fit. This suggests that the quadratic model, rather than being overly basic or complex, accurately captures the underlying patterns in the data. It successfully balances generalizing to fresh, untested data with fitting the training set. The link between the features and the goal variable is thought to be appropriately and effectively represented by this model

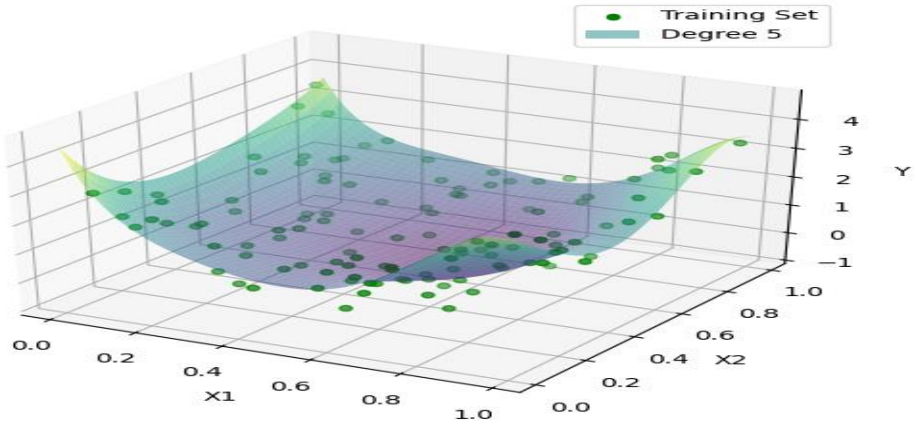
When degree=3 Balanced Fit

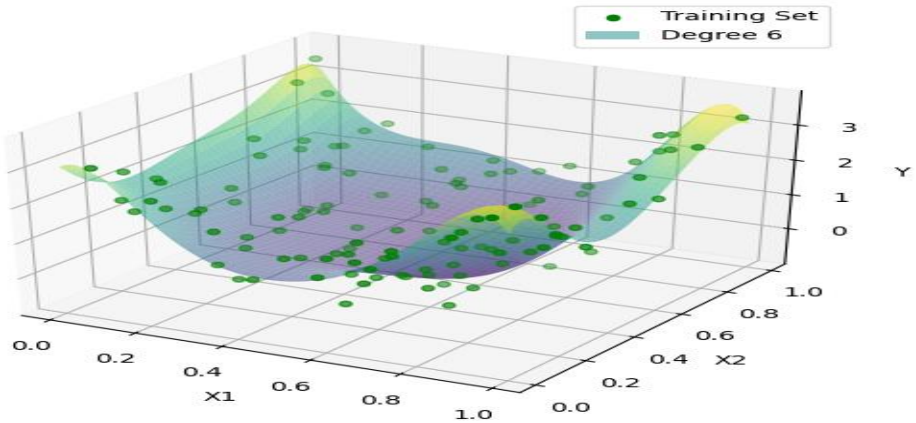


The third-order polynomial achieves an MSE value of 0.25, indicating a good balance between model complexity and fit. It captures the basic patterns effectively without being overly complex, and represents good patterns, but second-rate patterns are better.

When degree=4 , 5, 6 Balanced Fit



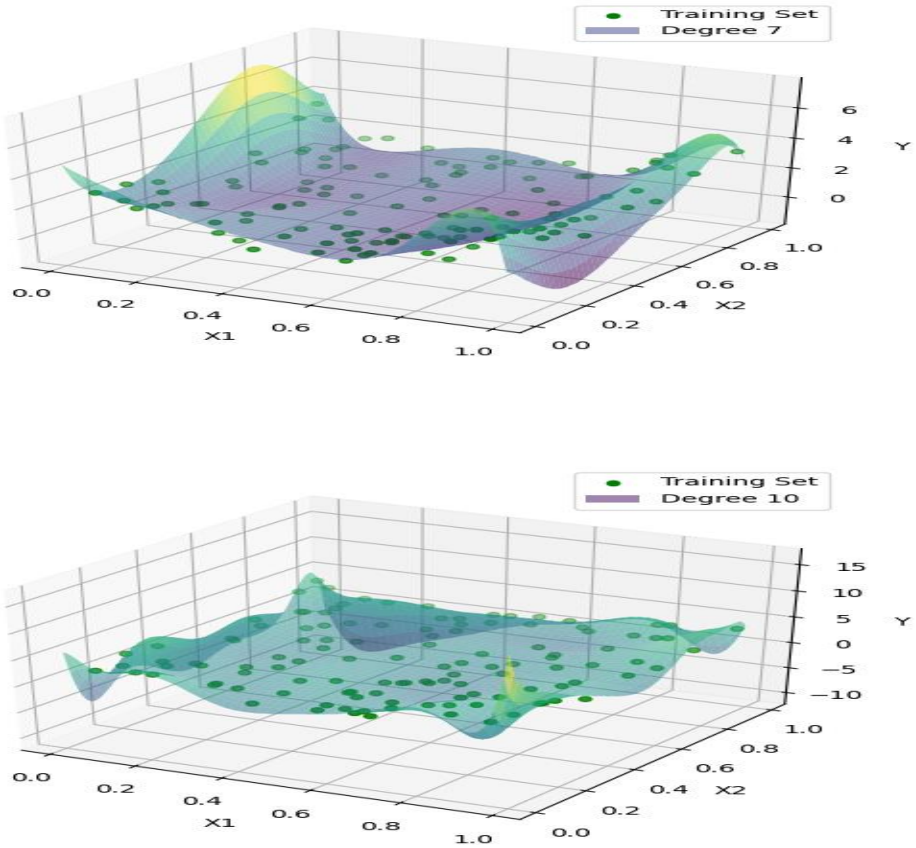




Degree 4, 5 and 6:

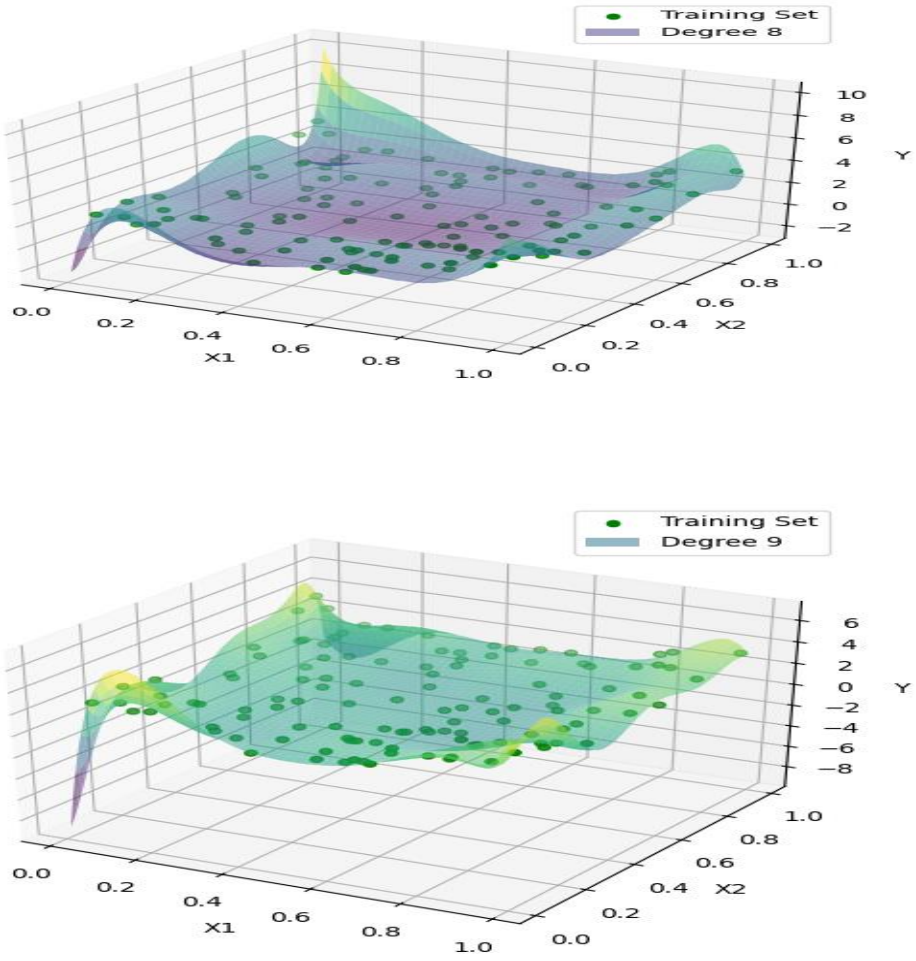
The fourth-, fifth- and sixth-order polynomials achieve an MSE value of approximately 0.3, indicating a good balance between model complexity and fit. It effectively captures basic patterns without being overly complex, which represents the best trade-off.

When degree=7 and degree=10 Overfitting



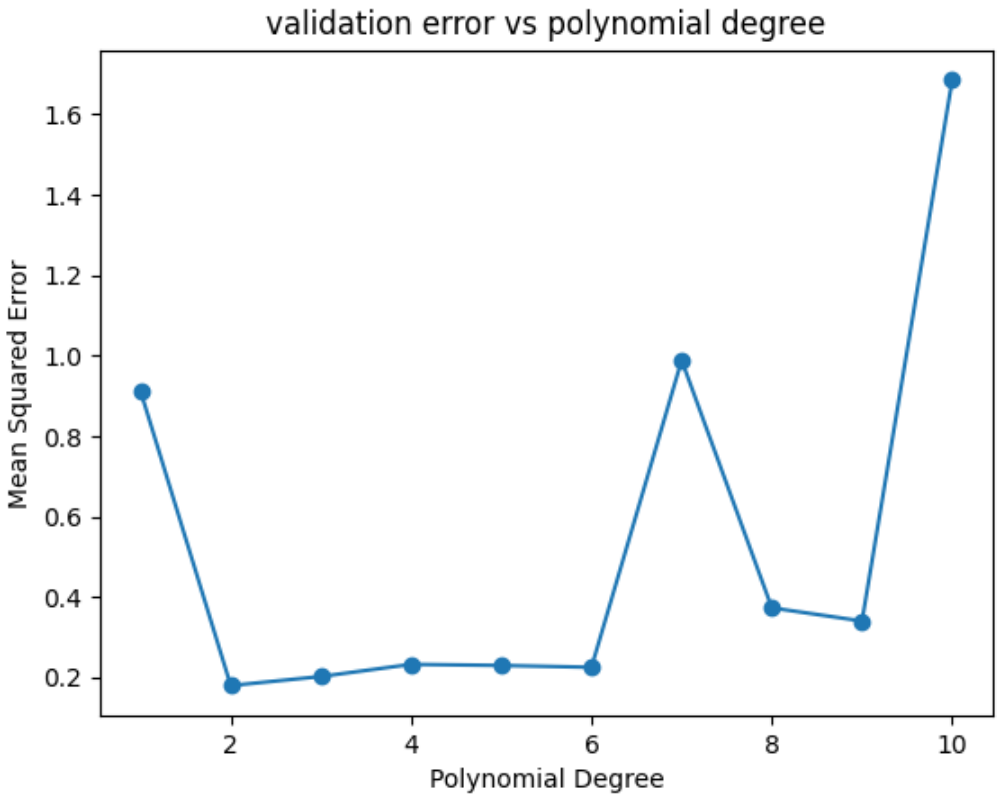
Overfitting (Degree 7 and Degree 10):Degree 7(MSE: 1.0) ,Degree 10(MSE: 1.8): The 7th-order and The 10th-order polynomial shows a significant increase in MSE to 1.0, indicating overfitting. This high error suggests that the model is fitting the training data too closely, capturing noise, and unlikely to generalize well to new data.

When degree=8,degree=9 *Overfitting*



the 8th and 9th-degree polynomials both exhibit signs of overfitting as reflected in their relatively low Mean Squared Errors (MSE) on the training data.

- *validation error vs polynomial degree curve*



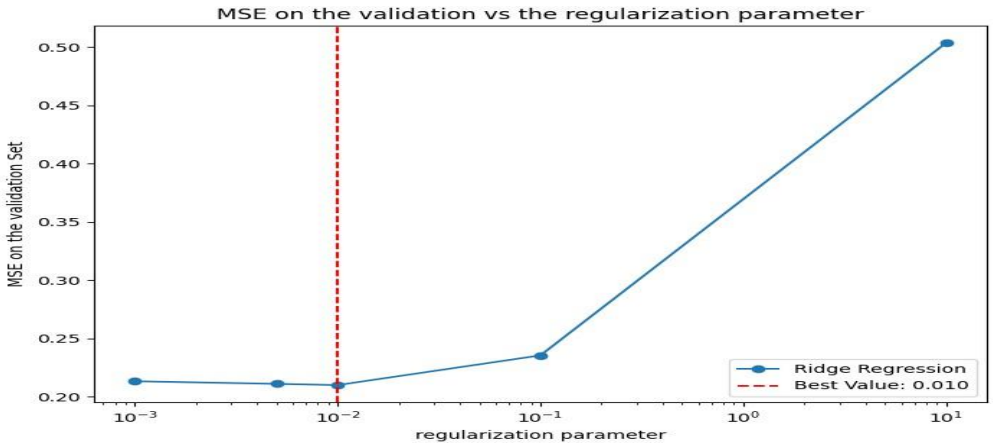
- Underfitting: Occurs when the model is too simple to capture the underlying patterns in the data.so the Result>>>> High bias, poor performance on both training and unseen data. (example when degree 1)
- Overfitting: Occurs when the model is overly complex, fitting the training data too closely.so the Result>>>>High variance, low performance on validation/unseen data. (example when degree 10)
- Best-Fit Model: Achieves an optimal balance between underfitting and overfitting.so the Result>>>> Generalizes well to new, unseen data. (example when degree 2)

Q1)3

Code:

```
# Question 1.3 Dana Bornata 1200284
ridge_Option = [0.001, 0.005, 0.01, 0.1, 10]
ridge_validationErrors = []
for alpha in ridge_Option:
    Polynomial = PolynomialFeatures(degree=8)
    X_train_Polynomial = Polynomial.fit_transform(Xtrainingset)
    Xvalidation_Polynomial = Polynomial.transform(Xvalidationset)
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_train_Polynomial, Ytrainingset)
    yvalidation_pred_ridge = ridge_model.predict(Xvalidation_Polynomial)
    val_error_ridge = mean_squared_error(Yvalidationset, yvalidation_pred_ridge)
    ridge_validationErrors.append(val_error_ridge)

best_value = ridge_Option[np.argmin(ridge_validationErrors)]
best_ridge_model = Ridge(alpha=best_value)
best_ridge_model.fit(X_train_Polynomial, Ytrainingset)
y_val_pred_best_ridge = best_ridge_model.predict(Xvalidation_Polynomial)

plt.figure(figsize=(8, 6))
plt.plot(ridge_Option, ridge_validationErrors, marker='o', label='Ridge Regression')
plt.axvline(x=best_value, color='r', linestyle='--', label=f'Best Value: {best_value:.3f}')
plt.xscale('log')
plt.title('MSE on the validation vs the regularization parameter')
plt.xlabel('regularization parameter')
plt.ylabel('MSE on the validation Set')
plt.legend()
plt.show()
```

Result:



Clarification:

The objective finds the optimal regularization parameter (alpha) that minimizes Mean Squared Error (MSE) on the validation set. GridSearchCV was employed to search for the best alpha value a set ([0.001, 0.005, 0.01, 0.1, 10]).

The chosen Ridge Regression model with alpha = 0.01 was considered the best model, providing a balanced fit to the training data while maintaining effectiveness in predicting new, unseen data.

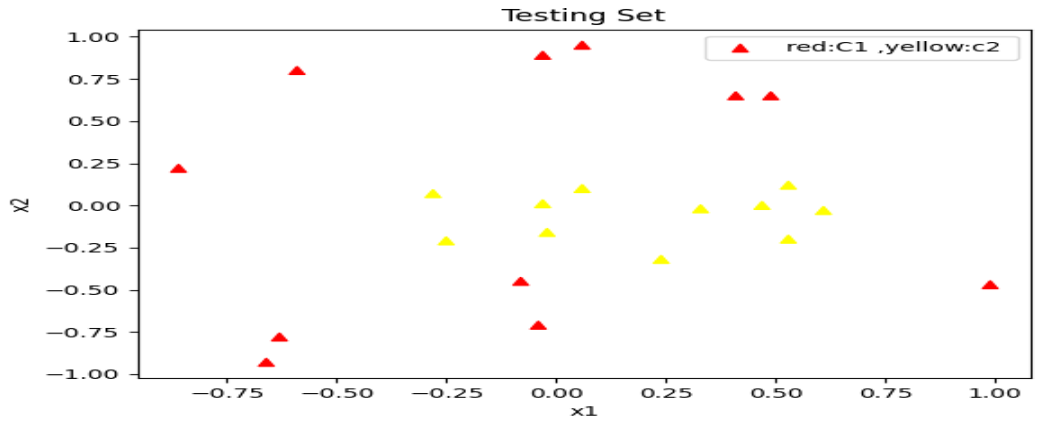
2) Logistic Regression

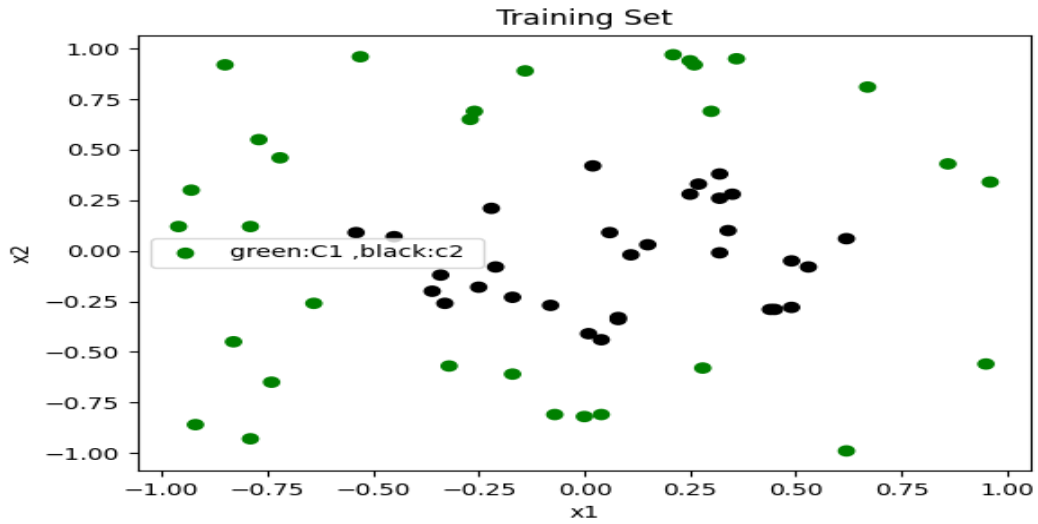
First: Representing the testing and training set

Code:

```python
# -*- coding: utf-8 -*-
import pandas as pd
import matplotlib.pyplot as plt


train_data = pd.read_csv('train_cls.csv')
test_data = pd.read_csv('test_cls.csv')

plt.scatter(train_data['x1'], train_data['x2'], c=train_data['class'].apply(lambda x: 'green' if x == 'C1' else 'bl
plt.title('Training Set')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()


plt.scatter(test_data['x1'], test_data['x2'], c=test_data['class'].apply(lambda x: 'red' if x == 'C1' else 'yellow'
plt.title('Testing Set')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

Result:





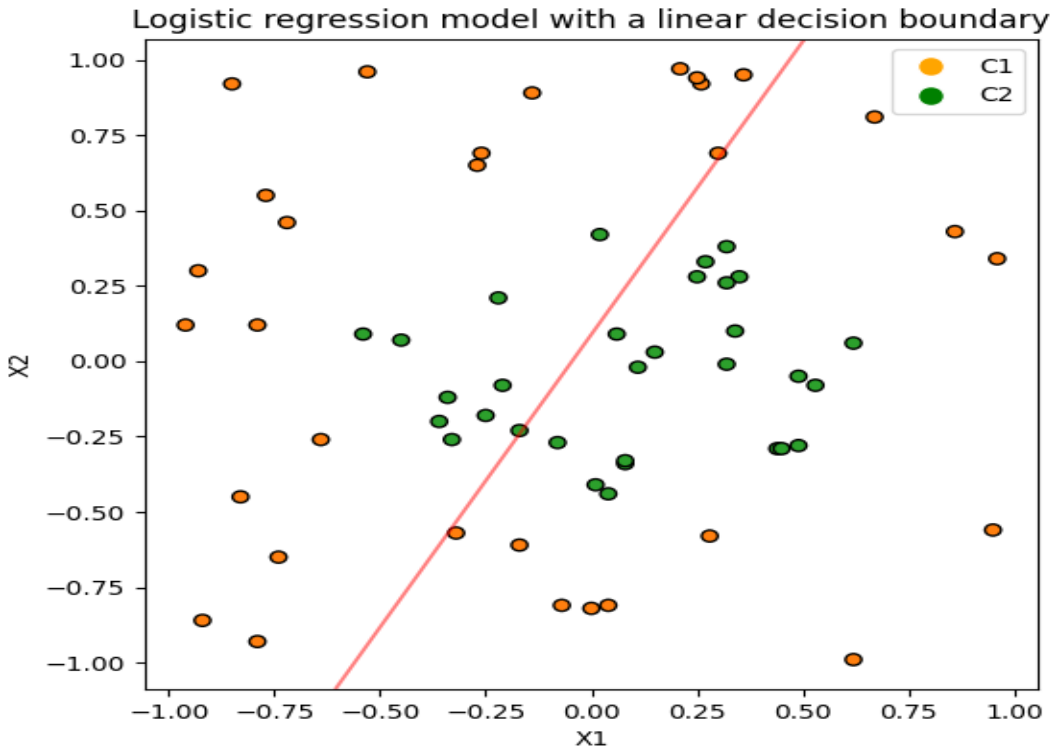
Q2)1

Code:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
# Q1-1 Dana 1200284

training_data = pd.read_csv('train_cls.csv')# Load the training data
testing_data = pd.read_csv('test_cls.csv') # Load the testing data
Xtraing = training_data[['x1', 'x2']].values
Ytraing = training_data['class'].values
Xtesting = testing_data [['x1', 'x2']].values
Ytesting = testing_data ['class'].values
# Initialize Logistic Regression model with linear decision boundary
linear_model = LogisticRegression()
linear_model.fit(Xtraing, Ytraing)
plt.figure(figsize=(6, 6))
plt.scatter(Xtraing[:, 0],Xtraing[:, 1], c=Ytraing, edgecolors='k', marker='o', label='Training Set')
plt.xlabel('X1')
plt.ylabel('X2')
# Plot decision boundary
a1 = plt.gca()
xlim = a1.get_xlim()
ylim = a1.get_ylim()
x, y = np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100))
Z = linear_model.decision_function(np.c_[x.ravel(), y.ravel()])
Z = Z.reshape(x.shape)
# Plot decision boundary
plt.contour(x, y, Z, colors='red', levels=[0], alpha=0.5, linestyles=['-'])
# Add legend for class labels
legend_labels = ['C1', 'C2']
```

```python
plt.xlabel('X1')
plt.ylabel('X2')
# Plot decision boundary
a1 = plt.gca()
xlim = a1.get_xlim()
ylim = a1.get_ylim()
x, y = np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100))
Z = linear_model.decision_function(np.c_[x.ravel(), y.ravel()])
Z = Z.reshape(x.shape)
# Plot decision boundary
plt.contour(x, y, Z, colors='red', levels=[0], alpha=0.5, linestyles=['-'])
# Add legend for class labels
legend_labels = ['C1', 'C2']
handles = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='orange', markersize=10),
           plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='green', markersize=10)]
plt.legend(handles=handles, labels=legend_labels)
plt.title('Logistic regression model with a linear decision boundary')
plt.show()
# Compute training accuracy
train_preds = linear_model.predict(Xtraing)
train_accuracy = accuracy_score(Ytraing, train_preds)
print(f'Training Accuracy (Linear): {train_accuracy}')
# Compute testing accuracy
test_preds = linear_model.predict(Xtraing)
test_accuracy = accuracy_score(Ytraing, test_preds)
print(f'Testing Accuracy (Linear): {test_accuracy}')
```

Result:



- **Training Accuracy (Linear): 0.6612903225806451**
- **Testing Accuracy (Linear): 0.6612903225806451**

Clarification:

The model achieves an accuracy of approximately 66.13% on both the training and testing datasets.

So, the training and testing accuracies are very close, suggesting that the model's performance on new data is consistent with its performance on the training data.

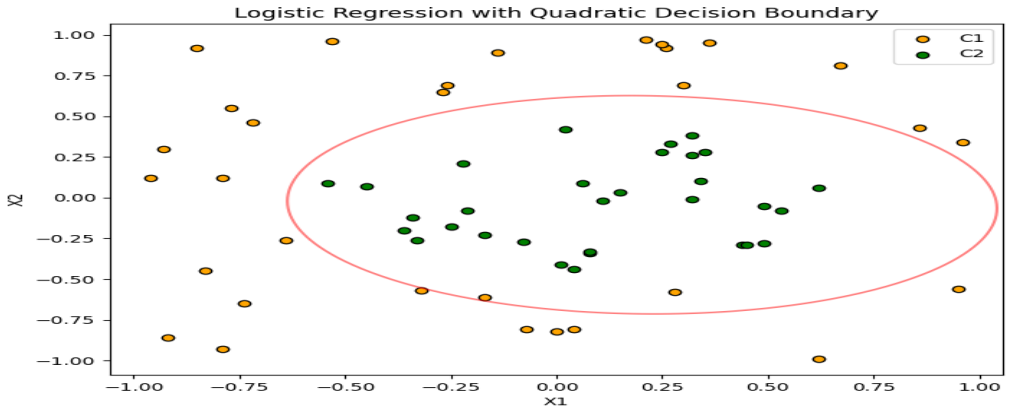
But This could suggest that the model is underfitting, or too basic, to fully capture the underlying patterns in the data. It performs poorly when it comes to generalizing the training and testing sets.

Q2)2

Code:

```
#Q2-2
# Initialize Logistic Regression model with quadratic decision boundary
quadratic_model = make_pipeline(PolynomialFeatures(degree=2), LogisticRegression())
quadratic_model.fit(Xtraing, Ytraing)
plt.figure(figsize=(8, 6))
plt.scatter(Xtraing[Ytraing == 'C1'][:, 0], Xtraing[Ytraing == 'C1'][:, 1], edgecolors='k', marker='o', c='orange', label='C1')
plt.scatter(Xtraing[Ytraing == 'C2'][:, 0], Xtraing[Ytraing == 'C2'][:, 1], edgecolors='k', marker='o', c='green', label='C2')
plt.xlabel('X1')
plt.ylabel('X2')
xlim, ylim = plt.gca().get_xlim(), plt.gca().get_ylim()
x, y = np.meshgrid(np.linspace(xlim[0], xlim[1], 100), np.linspace(ylim[0], ylim[1], 100))
Z = quadratic_model.decision_function(np.c_[x.ravel(), y.ravel()])
Z = Z.reshape(x.shape)
plt.contour(x, y, Z, colors=['red', 'blue'], levels=[0], alpha=0.5, linestyles=['-'])

# Add legend for class labels
plt.legend()
plt.title('Logistic Regression with Quadratic Decision Boundary')
plt.show()
# Compute training accuracy for quadratic model
train_preds_quad = quadratic_model.predict(Xtraing)
train_accuracy_quad = accuracy_score(Ytraing, train_preds_quad)
print(f'Training Accuracy (Quadratic): {train_accuracy_quad}')
# Compute testing accuracy for quadratic model
test_preds_quad = quadratic_model.predict(Xtraing)
test_accuracy_quad = accuracy_score(Ytraing, test_preds_quad)
print(f'Testing Accuracy (Quadratic): {test_accuracy_quad}')
```

Result:



Training Accuracy (Quadratic): 0.967741935483871

Testing Accuracy (Quadratic): 0.967741935483871

Clarification:

The findings suggest that training and testing accuracy are both reasonably good when utilising a logistic regression model with a quadratic decision boundary:

The quadratic training accuracy is 0.967741935483871.

The quadratic testing accuracy is 0.967741935483871.

High training and testing accuracies imply that the quadratic model is operating efficiently and making good generalizations to fresh, untested data.

Q2)3

Training Accuracy (Linear): 0.6612903225806451 in the Linear Model

Linear Testing Accuracy: 0.6612903225806451

Training Accuracy (Quadratic): 0.967741935483871 in the quadratic model

The quadratic testing accuracy is 0.967741935483871.

- Model Linear:

Training Accuracy: At roughly 66%, the training accuracy is not very high. This may indicate underfitting since it implies that the linear model is not sophisticated enough to fully represent the underlying patterns in the data.

Testing Accuracy: At roughly 66%, the testing accuracy is in line with the training accuracy. There is no discernible difference in the accuracies of testing and training, suggesting that overfitting has not occurred.

- The quadratic model

Training Accuracy: At approximately 97%, the training accuracy is high. This shows how well the quadratic model fitted the training set.

Testing Accuracy: At approximately 97%, the testing accuracy is likewise high, comparable to the training accuracy. Although this points to strong generalisation, there is a chance of overfitting, particularly when training and testing accuracies differ significantly.

So:

Given the complexity of the link between attributes and the target, it appears that the linear model underfits the data.

The quadratic model exhibits remarkable performance on both the training and testing datasets. Overfitting could occur, though, as the model might be too intricate for the available information.