Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS5341 Machine Learning and Data Science

Assignment #2

Prepared by:

Jana Herzallah 1201139

Instructor: Dr.Yazan Abufarha

Section: 1

Date: 21/12/2023

Contents

Table of figures:

Part 1: Model Selection and Hyper-parameters Tunning

The data_reg.csv file contains a set of 200 examples. Each row represents one example which has two attributes x1 and x2, and a continuous target label y. Using python, implement the solution of the following tasks:

1- Read the data from the csv file and split it into training set (the first 120 examples), validation set (the next 40 examples), and testing set (the last 40 examples). Plot the examples from the three sets in a scatter plot (each set encoded with a different color). Note that the plot here will be 3D plot where the x and y axes represent the x1 and x2 features, whereas the z-axis is the target label y.

Since data_reg.csv is data separated by a comma, I have chosen to use the pandas library because it has represents data in a tabular structure called dataframe.

This makes it easy to work with rows and columns of data, similar to how they are organized in a spreadsheet and it has lots of already built functions that we can easily call.

```python
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Step 1: Read data from the CSV file
data = pd.read_csv('data_reg.csv')
#since pandas have a really good data frames to manipulate data
#we can use it to read the data from the csv file especially if the data is in a table format

# Step 2: Split the data into training, validation, and testing sets
training_set = data[:120]#0-120 is the training set
validation_set = data[120:160] #120-160 is the validation set
testing_set = data[160:] #160-200 is the testing set

# Step 3: Create a 3D scatter plot
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
#111 means 1 row, 1 column, and the 1st plot it means that we only have 1 plot and in 3D mode

# Scatter plot for training set
ax.scatter(training_set['x1'], training_set['x2'], training_set['y'], c='r', marker='o', label='Training Set')

# Scatter plot for validation set
ax.scatter(validation_set['x1'], validation_set['x2'], validation_set['y'], c='g', marker='o', label='Validation Set')

# Scatter plot for testing set
ax.scatter(testing_set['x1'], testing_set['x2'], testing_set['y'], c='b', marker='^', label='Testing Set')
```

Figure 1 : scatter plot code

Then the data got separated to 3 sets; training 1-120 , validation 120-160 and training form 160 to 200. Then I have represented the training data with red dots, the validation data with green dots

and testing data with blue triangles to tell them apart. After that I have drawn a scatter plot that combines them all with having x,y axes x1 and x2 features and y have the z-axis as appers in figure2 below:

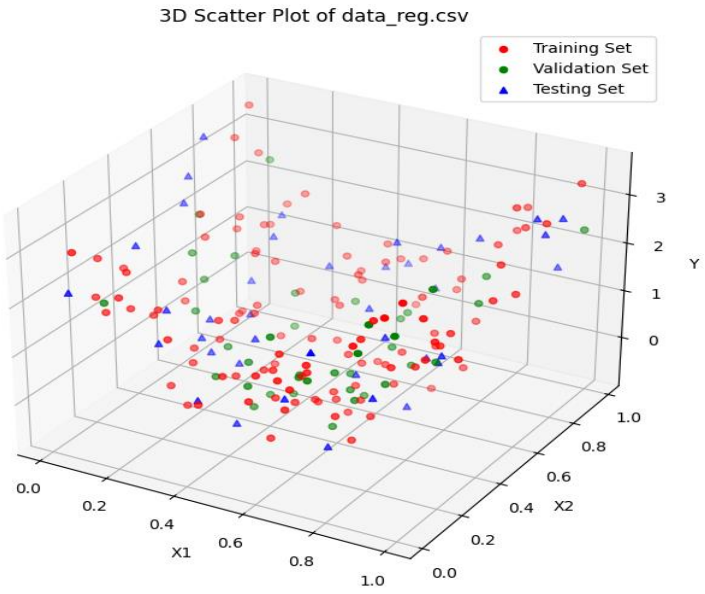

Figure 2 : 3D scatter plot of data

2- Apply polynomial regression on the training set with degrees in the range of 1 to 10. Which polynomial degree is the best? Justify your answer by plotting the validation error vs polynomial degree curve. For each model plot the surface of the learned function alongside with the training examples on the same plot. (hint: you can use PolynomialFeatures and LinearRegression from scikit-learn library)

```
features_train = training_set[['x1', 'x2']]
target_train = training_set['y']
features_validation = validation_set[['x1', 'x2']]
target_validation = validation_set['y']

# List to store validation errors for each degree
#i will use it to store the validation errors for each degree
validation_errors = []
```

Figure 3: preparing data for polynomial regression from degree 1 to degree 10

I have first started by preparing data including the training and validation sets to get polynomial regression but we have only 2 features originally (x1 and x2) so when there is a degree above 1, we need to use PolynomialFeatures to transform original data to the suitable data using monomials so for example for degree 2 we need to find x1^2 and x2^2 and x1x2

```
for degree in range(1, 11):
    # Create polynomial features function
    poly_features = PolynomialFeatures(degree)
    #degree is the degree of the polynomial function
    #we will use the polynomial function to transform the original features to polynomial features with the degree we want

    # Transform training and validation features to fit in the polynomial function of the specified degree
    X_train_poly = poly_features.fit_transform(features_train)
    X_validation_poly = poly_features.transform(features_validation)

    # Initialize linear regression model
    regression_model = LinearRegression(fit_intercept=True)
    #fit_intercept=True means that we will have a bias
    # Train the model
    regression_model.fit(X_train_poly, target_train)
    #fit is a function that will train the model regarding input data and target data
    # Predict on the validation set
    YpredictedValidation = regression_model.predict(X_validation_poly)

    # Calculate mean squared error
    validation_error = mean_squared_error(target_validation, YpredictedValidation)
    # mean_squared_error is a function that will calculate the mean squared error between predicted and actual values
    # target_validation is the actual values and y_val_pred is the predicted values
    # we will store the validation error in the validation_errors list
    validation_errors.append(validation_error)
    plot_surface_with_degree(regression_model, poly_features, training_set, validation_set, degree)
e in range(1, 11)
```

Figure 4 : looping for each degree to create polynomials regression

Here I did a few things to understand how well polynomial regression works on a dataset. I began by extracting the features (represented by x1 and x2) and the target variable (y) from training and validation sets. I then sat up an empty list called "validation_error" to keep track of the errors for each degree of the polynomial regression. In the running for loop from 1 to 10 degrees, in each iteration it creates polynomial features needed regarding to the degree and training a model for

each degree. And then it computes the mean squared error between the predicted values and the actual values in the validation set then it saves these errors in the list. Additionally, I visualized the learned functions by plotting their surfaces in 3D alongside the training data using plot surface with degree function that I built above which I will talk about below . Finally, i created a plot showing how the validation error changes with different degrees, to find out the least validation error computed by which degree.

```
99  def plot_surface_with_degree(regression_model, poly_features, training_set, validation_set, degree):
100     fig = plt.figure(figsize=(8, 8))
101     ax = fig.add_subplot(111, projection='3d')
102
103     x1_mesh, x2_mesh = generate_meshgrid(training_set)
104
105     predicted_y = transform_and_predict(regression_model, poly_features, x1_mesh, x2_mesh)
106
107     plot_surface_and_points(ax, x1_mesh, x2_mesh, predicted_y, training_set, validation_set, degree)
108
109     plt.show()
110
```

Figure 5 : plot surface with degree function

This function was made up of 3 main components which are generate meshgrid , transform and predict and lastly plot surface and points

And I will discuss each one below:

```
def generate_meshgrid(features_train, num_points=100):
    x1_range = np.linspace(features_train['x1'].min(), features_train['x1'].max(), num_points)
    x2_range = np.linspace(features_train['x2'].min(), features_train['x2'].max(), num_points)
    x1_mesh, x2_mesh = np.meshgrid(x1_range, x2_range)
    return x1_mesh, x2_mesh
#meshgrid is a function that will create a grid of points
#we will use it to create a grid of points for the x1 and x2 features
#we will use the grid of points to plot the surface of the polynomial function
```

Figure 6 : generate_meshgrid function

For this part of the code, I have done these steps in order to plot each surface for each polynomial degree on 3d plot. I began by determining the ranges for the features x1 and x2 usingg NumPy's linspace functions, then I used meshgrid function to create a grid of coordinates over these ranges of x1 and x2.

```
76  def transform_and_predict(regression_model, poly_features, x1_mesh, x2_mesh):
77      new_data = poly_features.transform(np.c_[x1_mesh.ravel(), x2_mesh.ravel()])
78      predicted_y = regression_model.predict(new_data).reshape(x2_mesh.shape)
79      return predicted_y
80  #transform is a function that will transform the original features to polynomial features
81  #we will use it to transform the original features to polynomial features with the degree we want
82  #we will use the polynomial features to train the model and predict the target values and to plot the surface of the polynomial function
83
```

Figure 7 : transform and predict function

Then the polynomial features are transformed for each coordinate pair, and the target variable is predicted using a previously trained polynomial regression model.

```python
def plot_surface_and_points(ax, x1_mesh, x2_mesh, predicted_y, training_set, validation_set, degree):
    ax.scatter(training_set['x1'], training_set['x2'], training_set['y'], c='r', marker='o', label='Training Set')
    ax.scatter(validation_set['x1'], validation_set['x2'], validation_set['y'], c='g', marker='o', label='Validation Set')
    ax.plot_surface(x1_mesh, x2_mesh, predicted_y, alpha=0.5, cmap='viridis', label=f'Ploynomial Degree {degree} Surface')
    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    ax.set_zlabel('Y')
    ax.legend()
    ax.set_title(f'Degree {degree}')
#plot_surface is a function that will plot the surface of the polynomial function
#we will use it to plot the surface of the polynomial function of the specified degree
#we will use the polynomial function to transform the original features to polynomial features with the degree we want
```

Figure 8 : plot surface and points function

Then for each polynomial degree a 3D surface was plotted which represents the learned polynomial surface, while scattered points depict the training set and the validating data sets.

And these are the results obtained for each degree:

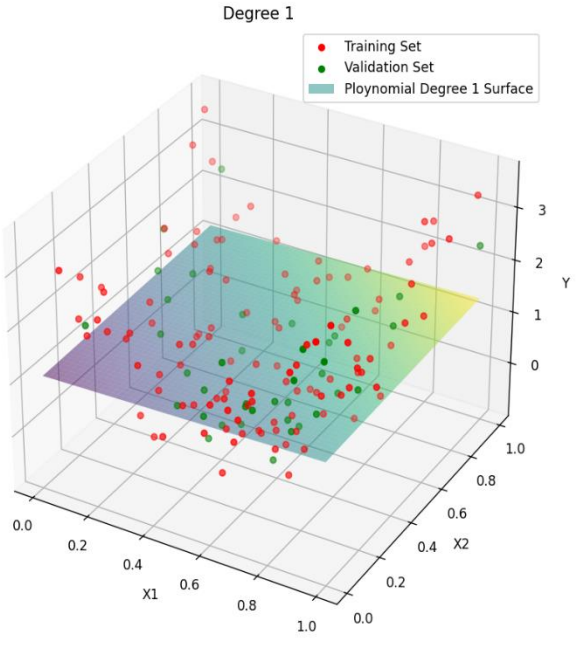

Figure 9 :polynomial regression of degree 1

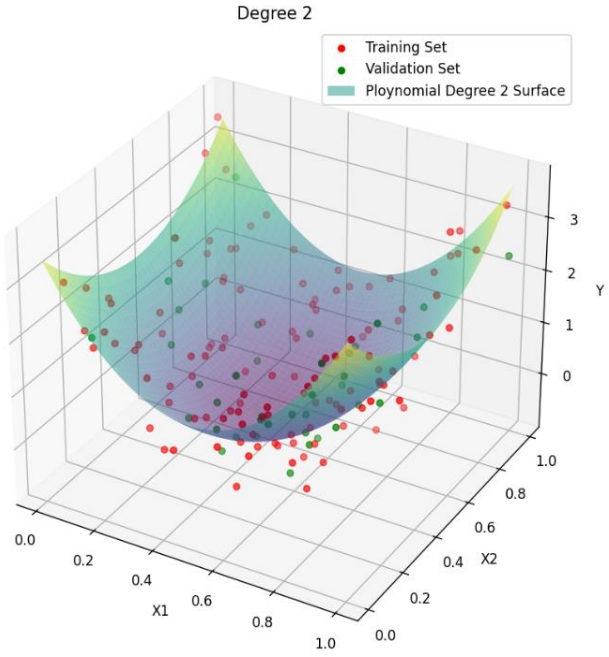

Figure 10: polynomial regression of degree 2

Its noticeable that validation set lies better on the polynomial of degree2 than degree 1.

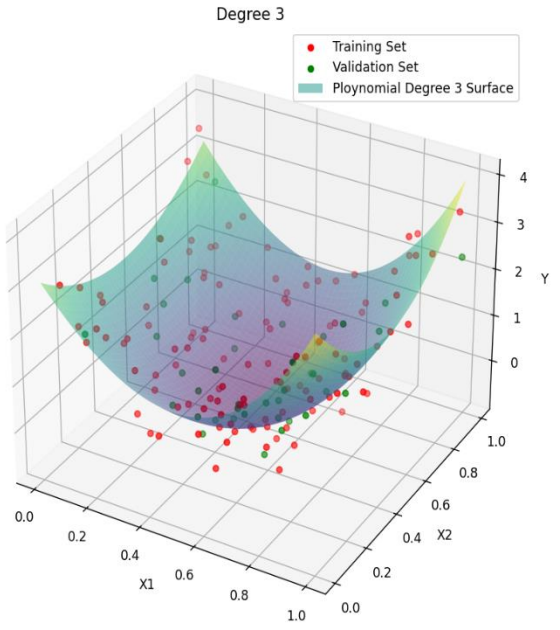

Figure 12 :polynomial regression of degree 3

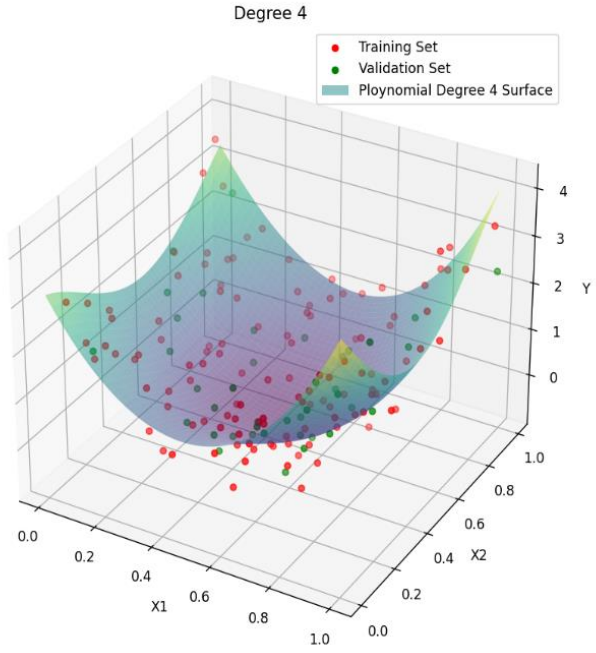

Figure 11 :polynomial regression of degree 4

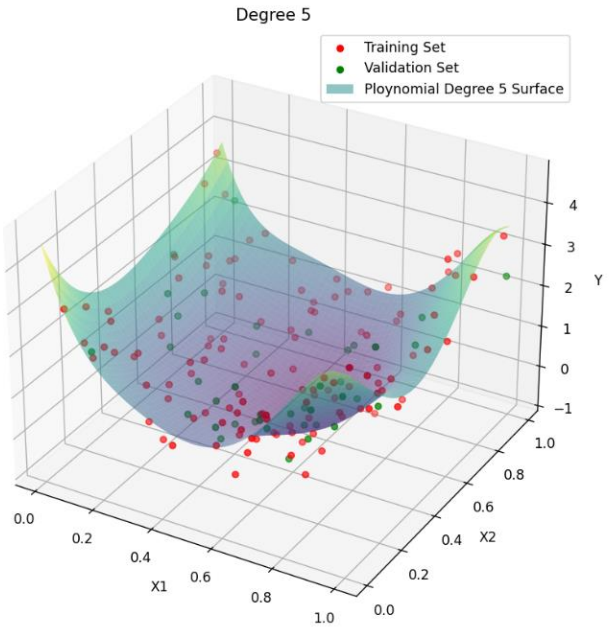

Figure 14:polynomial regression of degree 5

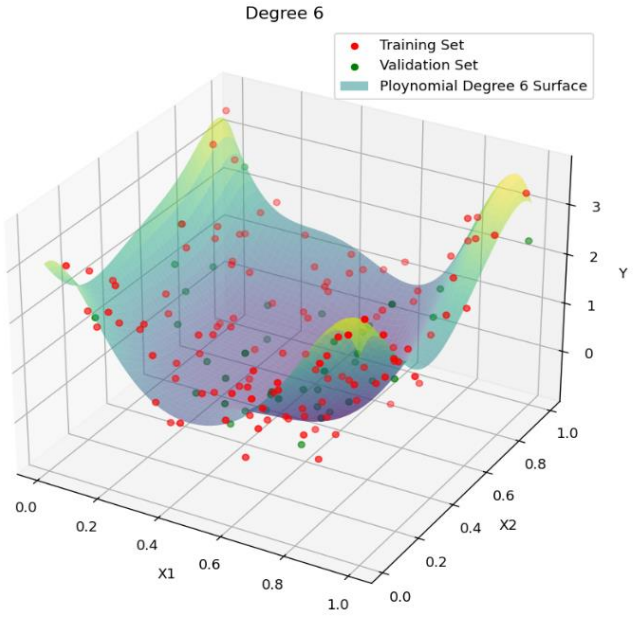

Figure 13:polynomial regression of degree 6

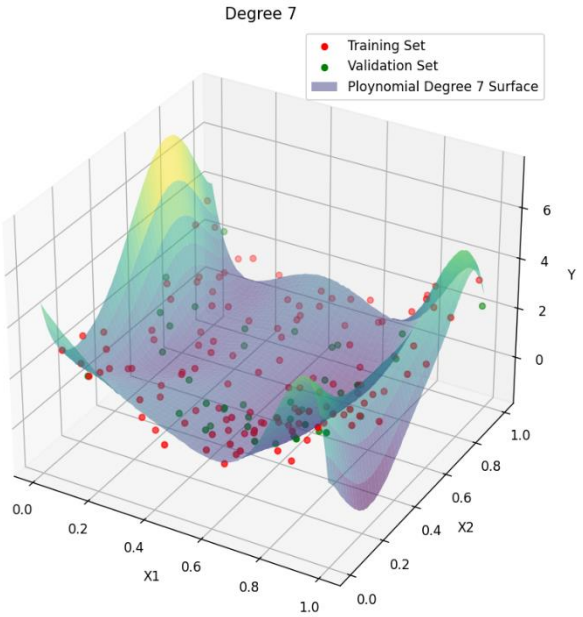

Figure 16 : polynomial regression of degree 7

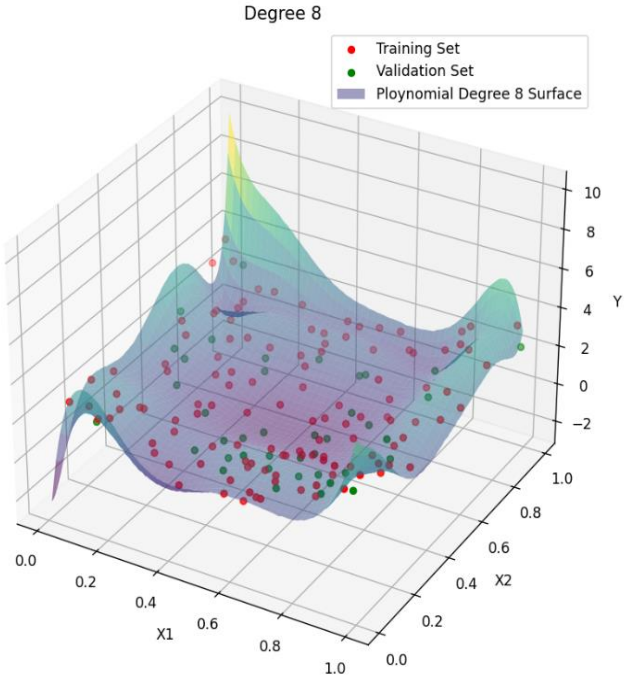

Figure 15 :polynomial regression of degree 8

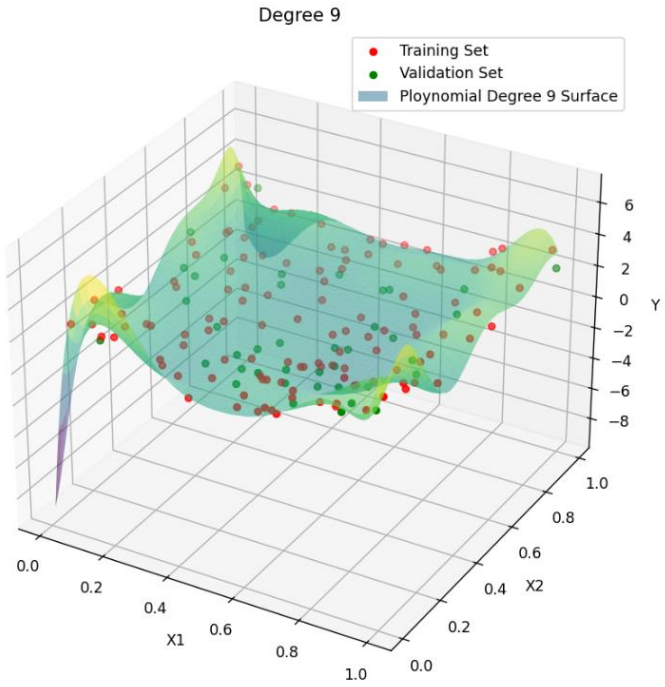

Figure 17:polynomial regression of degree 9

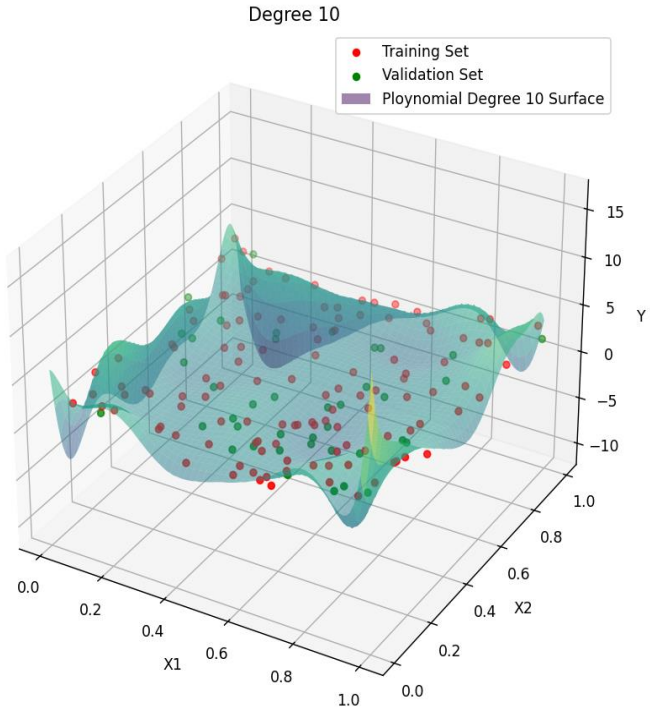

Figure 18:polynomial regression of degree 10

After plotting all polynomials from degree 1 to 10 , degree 2 happedned to be the least mean squared error between the predicted valuse and true values when applied on validation sets.

Figure 19 : validation sets on all degrees

Degree 2 has the least validation error so it's the best out of them and its obvious that the surface of degree 2 fits best with validation set between all other surfaces

```
# Print validation errors for each degree
for degree, error in zip(range(1, 11), validation_errors):
    print(f'Degree {degree}: Validation Error = {error}')

# Determine the degree with the least error
best_degree = np.argmin(validation_errors) + 1  # Add 1 to convert index to degree
best_error = min(validation_errors)
print(f'The best polynomial degree is: {best_degree} with a validation error of {best_error}')

# Plot validation error vs polynomial degree
plt.figure(figsize=(6, 6))
plt.plot(range(1, 11), validation_errors, marker='o', color='green')
plt.xlabel('Polynomials Degree')
plt.ylabel('Validations Error')
plt.title('Validation Error vs. Polynomial Degree')
plt.show()
```

Figure 20 : code of printing validation errors and determining the least one of them

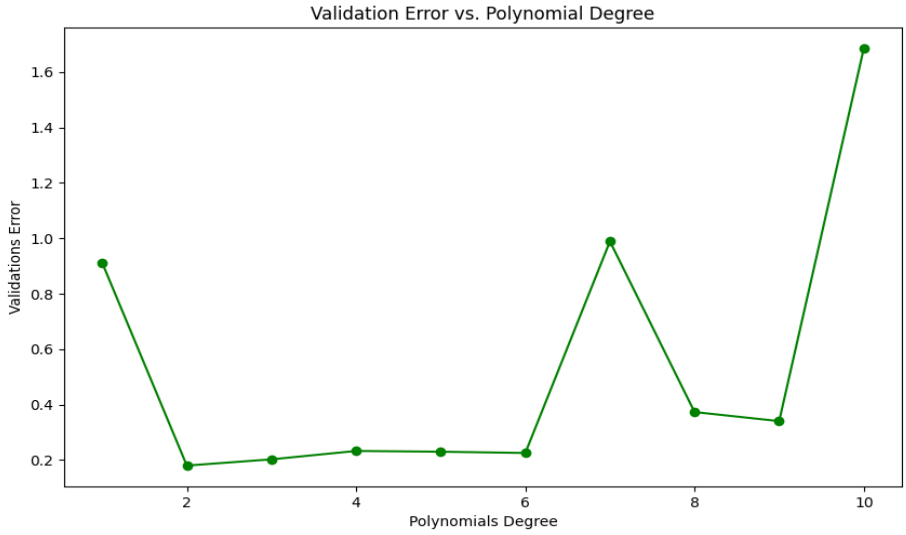

Figure 21 : validation error vs polynomial degree

Its obvious that degree 2 has the least validation error according to the graph in figure 18

3- Apply ridge regression on the training set to fit a polynomial of degree 8.

For the regularization parameter, choose the best value among the following options: {0.001, 0.005, 0.01, 0.1, 10}. Plot the MSE on the validation vs the regularization parameter. (hint: you can use Ridge regression implementation from scikit-learn)

To choose the best regularization parameter, I looked for the value that minimizes the Mean Squared Error (MSE) on the validation set. I tried regularization parameters provided {0.001, 0.005, 0.01, 0.1, 10}, and calculated the corresponding MSE values and then they were plotted as below:

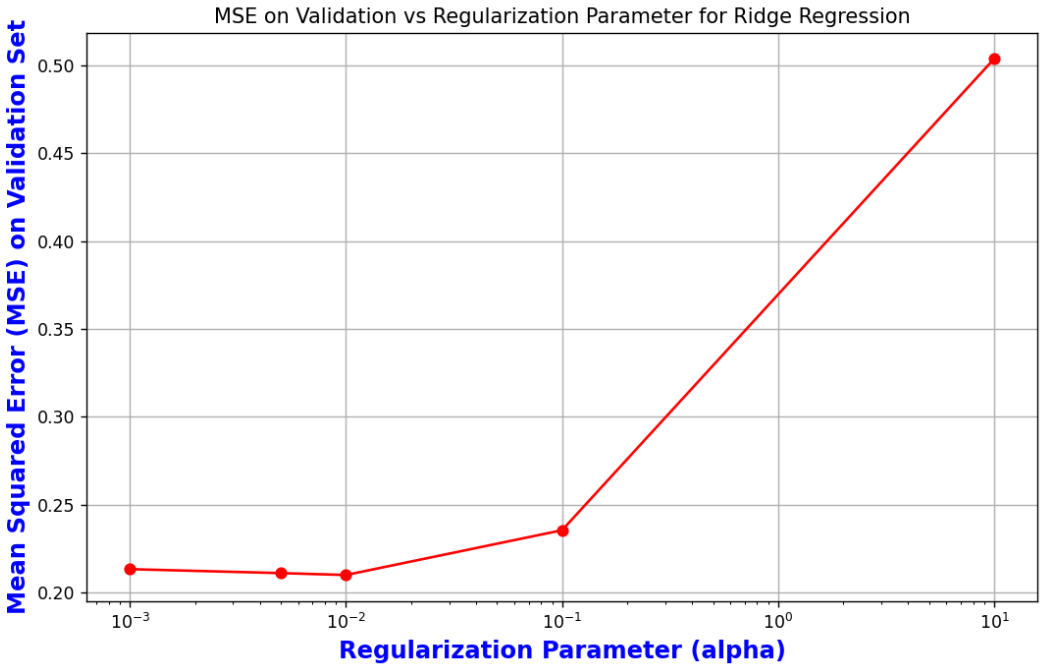

Figure 22 : plot of MSE on validation set vs regularization parameter

```
MSE for each regularization parameter:
alpha=0.001: MSE=0.21328335916736343
alpha=0.005: MSE=0.2110332820264636
alpha=0.01: MSE=0.20996554038850396
alpha=0.1: MSE=0.23545304328552383
alpha=10: MSE=0.5038254258404573
The best regularization parameter is: 0.01 with a MSE of 0.20996554038850396
```

Figure 23 : calculation of MSE for each alpha value

Its obvious that the least MSE was 0.20996554 which was corresponding to alpha = 0.01.

```python
degree = 8
poly = PolynomialFeatures(degree=degree)

X_poly_train = poly.fit_transform(training_set[['x1', 'x2']])
X_poly_val = poly.transform(validation_set[['x1', 'x2']])

# Regularization parameters to try
alpha_values = [0.001, 0.005, 0.01, 0.1, 10]
mse_values = []

for alpha in alpha_values:
    # Fit Ridge regression model
    ridge_model = Ridge(alpha=alpha)
    ridge_model.fit(X_poly_train, training_set['y'])

    # Predict on validation set
    y_val_pred = ridge_model.predict(X_poly_val)

    # Calculate Mean Squared Error
    mse = mean_squared_error(validation_set['y'], y_val_pred)
    mse_values.append(mse)
#mean_squared_error is a function that will calculate the mean squared error between predicted and actual values
#print the mse_values

# Print MSE for each regularization parameter
print('MSE for each regularization parameter:')
for alpha, mse in zip(alpha_values, mse_values):
    print(f'alpha={alpha}: MSE={mse}')
```

Figure 24 : code of ridge regression on the training set to fit a polynomial of degree 8

First ridge regression was employed to fit a polynomial of degree 8 on a given training set, and then the validation set was used to evaluate the the model's performance. The features of both sets were transformed using PolynomialFeatures from scikit-learn to fit in with degree 8.

For each parameter, a Ridge regression model was trained using training data, then predictions were made on the validation set (y was found from x features), and the Mean Squared Error (MSE) was calculated. The MSE values for each regularization parameter were printed and plotted.

```python
    mse = mean_squared_error(validation_set['y'], y_val_pred)
    mse_values.append(mse)
#mean_squared_error is a function that will calculate the mean squared error between predicted and actual values
#print the mse_values

# Print MSE for each regularization parameter
print('MSE for each regularization parameter:')
for alpha, mse in zip(alpha_values, mse_values):
    print(f'alpha={alpha}: MSE={mse}')

# Determine the regularization parameter with the least error
best_alpha = alpha_values[np.argmin(mse_values)]
best_mse = min(mse_values)
print(f'The best regularization parameter is: {best_alpha} with a MSE of {best_mse}')
```

Figure 25: calculating mse for each alpha value then print them with determining the least

Part 2: Logistic Regression

The train_cls.csv file contains a set of training examples for a binary classification problem, and the testing examples are provided in the test_cls.csv file. The following figures show these examples.

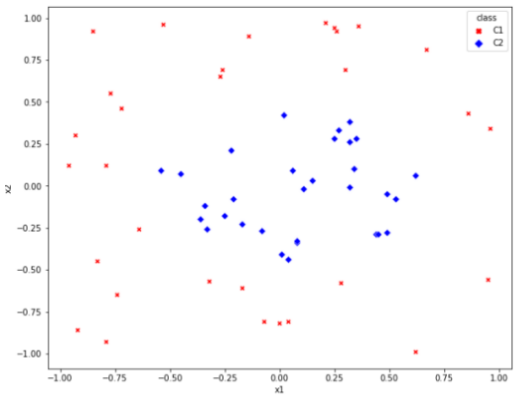

Figure 1 Training Set

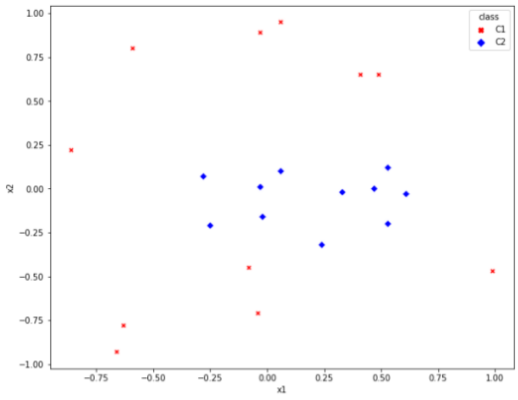

Figure 2 Testing Set

1. using the logistic regression implementation of scikit-learn library, Learn a logistic regression model with a linear decision boundary. Draw the decision boundary of the learned model on a scatterplot of the training set (similar to Figure 1). Compute the training and testing accuracy of the learned model.

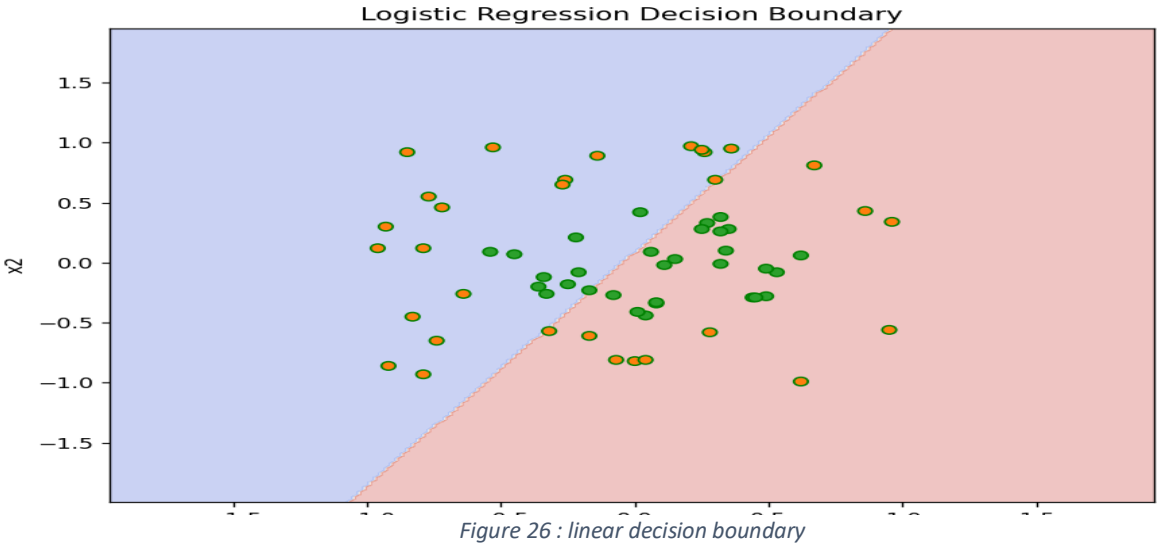

Figure 26 : linear decision boundary

```
plt.scatter(X_train['x1'], X_train['x2'], c=y_train, edgecolors='g', cmap=plt.cm.Pai
Training Accuracy: 0.6612903225806451
Testing Accuracy: 0.6818181818181818
```

Figure 27 : testing and training accuracy linear boundary

2. Repeat part 1 but now to learn a logistic regression model with quadratic decision boundary.

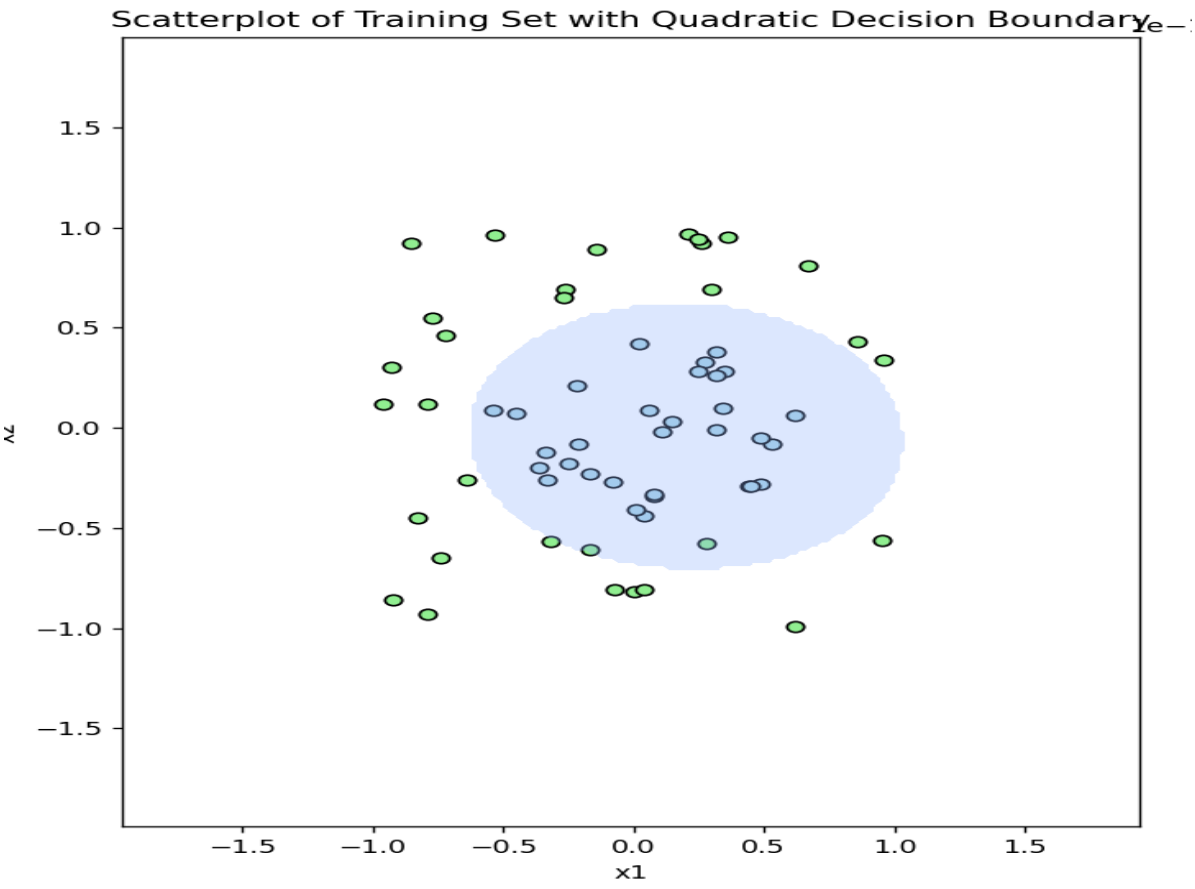

Figure 28 : quadratic decision boundary

```
Training Accuracy (Quadratic): 96.77%
Testing Accuracy (Quadratic): 95.45%
```

Figure 29 : training and testing accuracy quadratic decision boundary

3. Comment on the learned models in 1 and 2 in terms of overfitting/underfitting

For the linear decision boundary, its obvious that the line doesn't really tell apart in a correct way between the two classes (c1 and c2) and it really messed with some testing examples and training examples too and that gives an indication that the model is too simple and can't classify in an efficient way meaning that its an underfitting model ; if we look at the accuracies of training and testing data we can see that it got only 66% of them classified right which is not enough because the model could be a sensitive system such as diagnosing a disease or installed in a self-driving car.

While in the quadratic decision boundary, the accuracy in both training and testing data was really high and the gap between them was less than 2% which gives a really good indication of the model. Its obvious in the figure 28 how the contour nearly classified each point correctly and it only missed very little amount of them and that says that this model is a good fit.

References:

https://data36.com/polynomial-regression-python-scikit-learn/

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html