**Faculty of Engineering & Technology**

**Electrical & Computer Engineering Department**

**Machine Learning and Data Science - ENCS5341**

**Assignment #2**

---

**Prepared by:**

Ahmed Zubaidia 1200105

**Instructor:** Dr. Yazan Abu Farha

**Section:** 2

**Date:** 20/12/2023

# Table of Contents

*Table of Figures:*

# Solution:

## Part1

**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.

*Figure 1: part 1 text*

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression, Ridge, LogisticRegression
from sklearn.metrics import mean_squared_error, accuracy_score
```

*Figure 2: libraries that I used.*

At first we will read the data, and slice it to three sets, train data with 120 sample , validation with 40 samples , and test set with 40 samples also.

```python
# read the data

data = pd.read_csv('data_reg.csv')  data: ['x1', 'x2', 'y'] [0    0.548814  0.311796  0.547818] [1    0.715189  0.696343  0.576032] [2    0.602763  0.377752  0.113475] [3

# split the data into train and test and validation

train_data = data.iloc[:120] # train data from 0 to 120  train_data: ['x1', 'x2', 'y'] [0    0.548814  0.311796  0.547818] [1    0.715189  0.696343  0.576032] [2    0.6027

validation_data = data.iloc[120:160] # validation data from 120 to 160  validation_data: ['x1', 'x2', 'y'] [120  0.725254  0.270328  0.486485] [121  0.501324  0.131483  0.8

test_data = data.iloc[160:] # test data from 160 to the end  test_data: ['x1', 'x2', 'y'] [160  0.697429  0.187131 -0.265093] [161  0.453543  0.903984  0.963340] [162  0.72
```

*Figure 3: code which I used to make train , validation, test sets.*

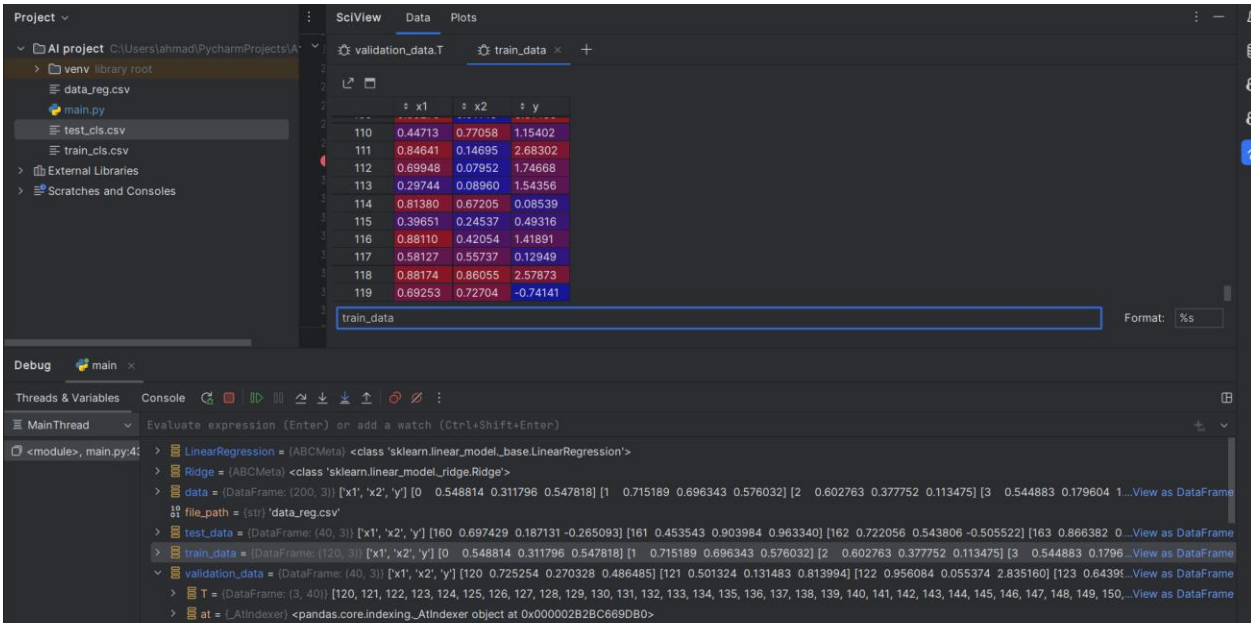


*Figure 4: train data 120 sample*

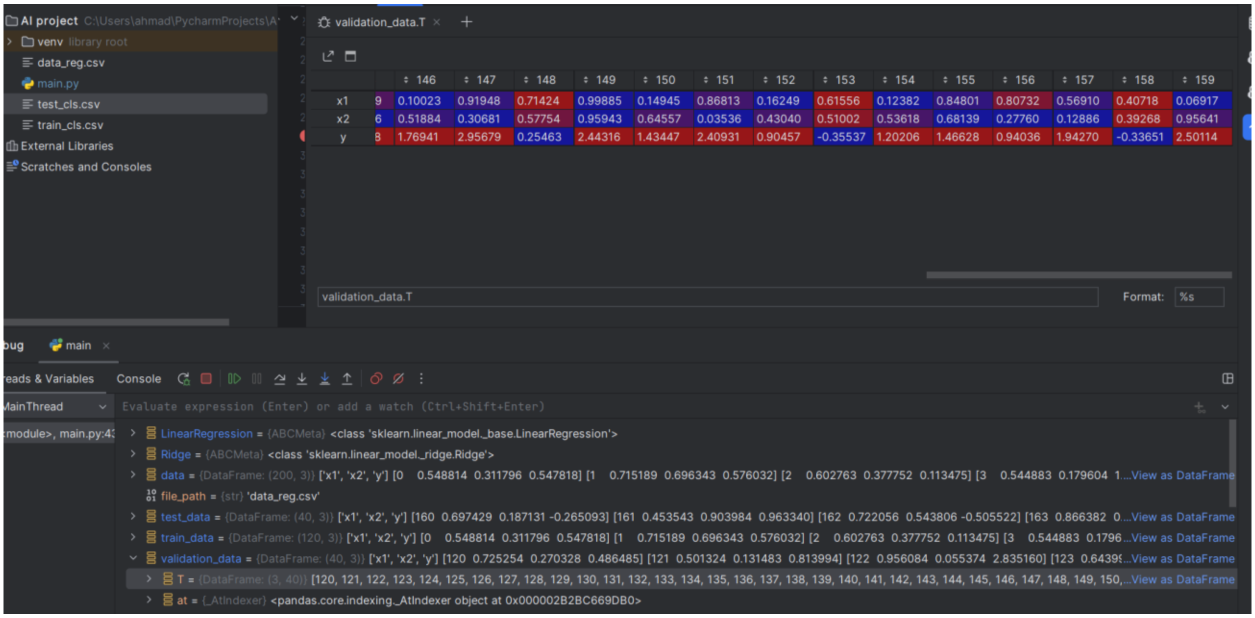


*Figure 5: validation set 40 samples.*

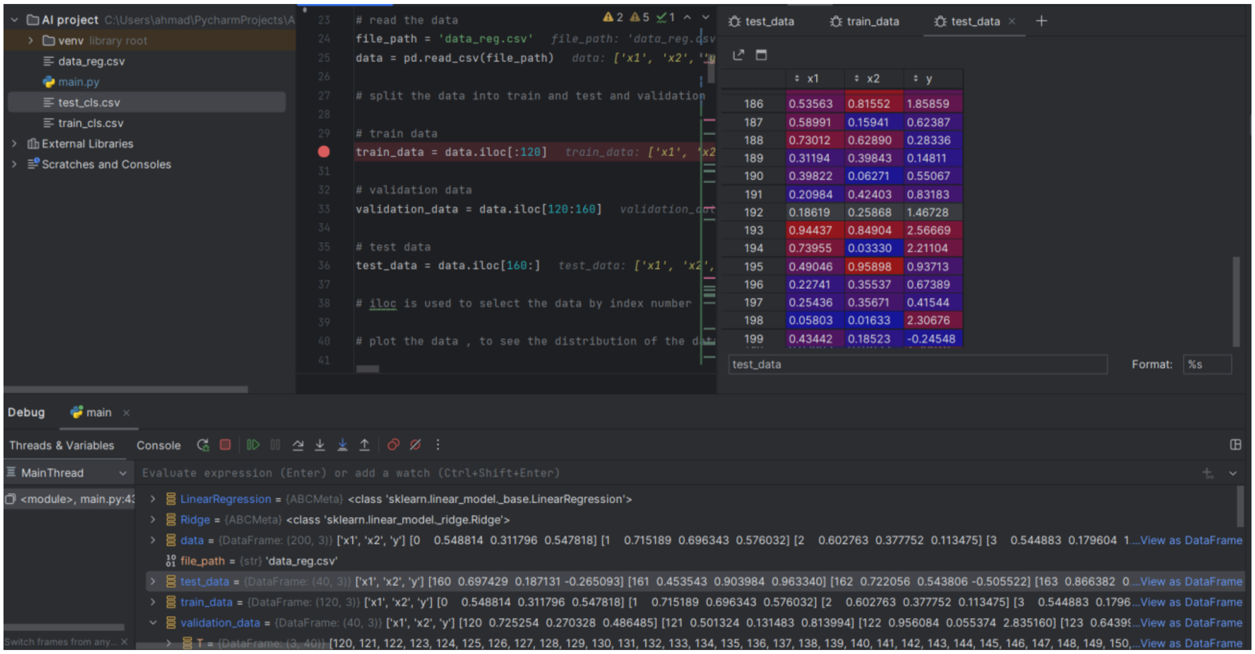



*Figure 6:test set 40 samples.*

Now we will plot these three sets with different colors, in 3D plane as required:

First, I create an object with (10,8) width and height, to make the plots on it, then telling the object to make sub plot which takes all the space of the object and make it in 3d plane.

Red for training data, green for validation, and blue for testing data.

```python
# ploting the data in 3D

fig = plt.figure(figsize=(10, 8))  # Create a figure object with size 10x8 inches.
ax = fig.add_subplot(111, projection='3d')

# Plot each dataset with a different color
ax.scatter(train_data['x1'], train_data['x2'], train_data['y'], color='r', label='Training Set')  # * Plot the training data red

ax.scatter(validation_data['x1'], validation_data['x2'], validation_data['y'], color='g', label='Validation Set')  # * Plot the validation data green

ax.scatter(test_data['x1'], test_data['x2'], test_data['y'], color='b', label='Testing Set')  # * Plot the testing data blue

# Labeling
ax.set_xlabel('X1')
ax.set_ylabel('X2')
ax.set_zlabel('Y')
ax.set_title('3D Scatter Plot of the Data')
ax.legend()  # Show the legend in the plot (the labels of the datasets)

# Show the plot
plt.show()
```

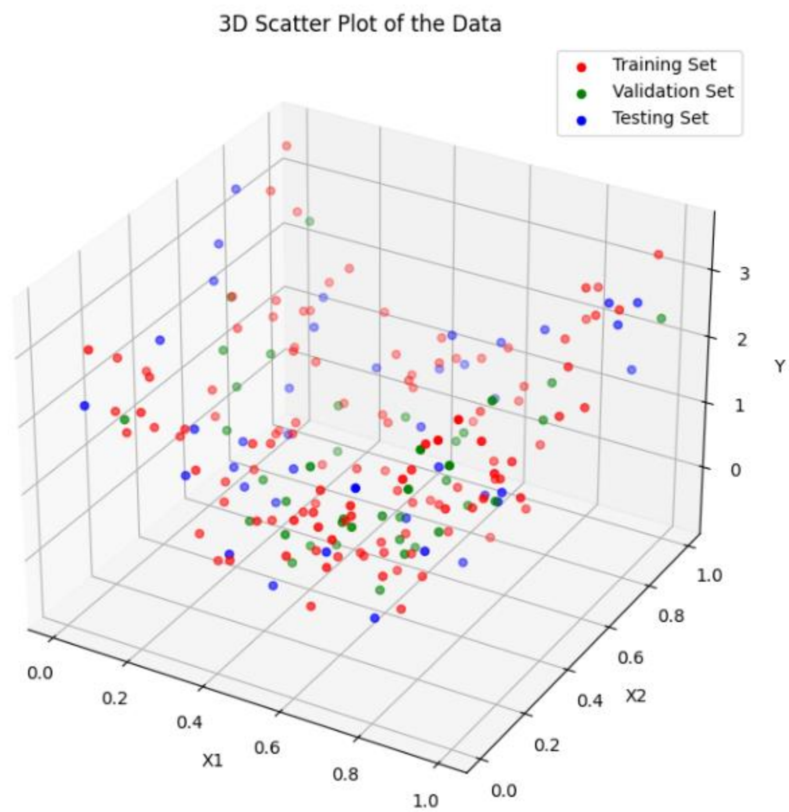



*Figure 7: plot of data sets in 3D*

## Part 2

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)

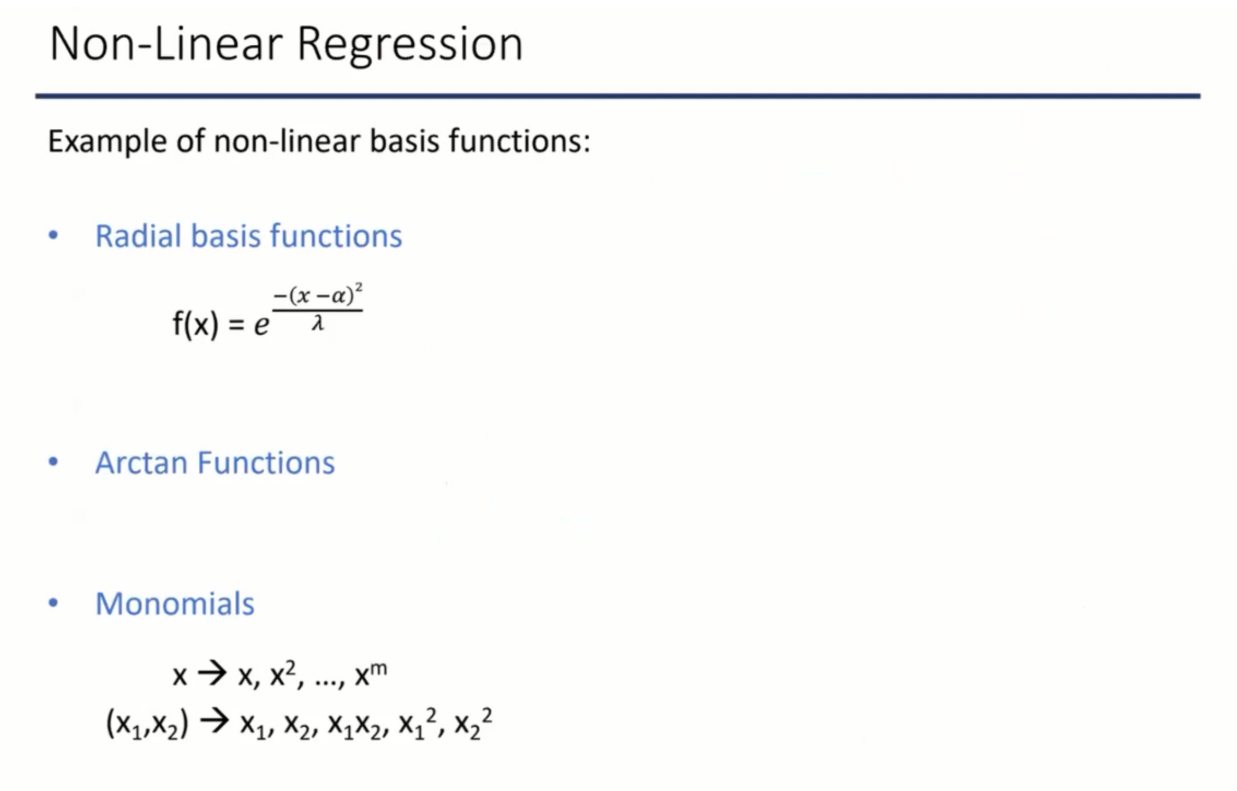



*Figure 8: universal basis function*

In this section , I used in my code the monomials to generate features according to each degree from 1 to 10 , by using the function **PolynomialFeatures(),**for example if the degree is 2 and the original features are **X** =(x1,x2) the polynomial function will generate Xpoly = [1, x1 ,x2 ,(x1)^2 ,(x2)^2, x1 * x2 ] features. Now in my code also when the Xpoly generate the features functions , I used it to substitute the values of that features and get the final result . for example here is snapshot of my code :

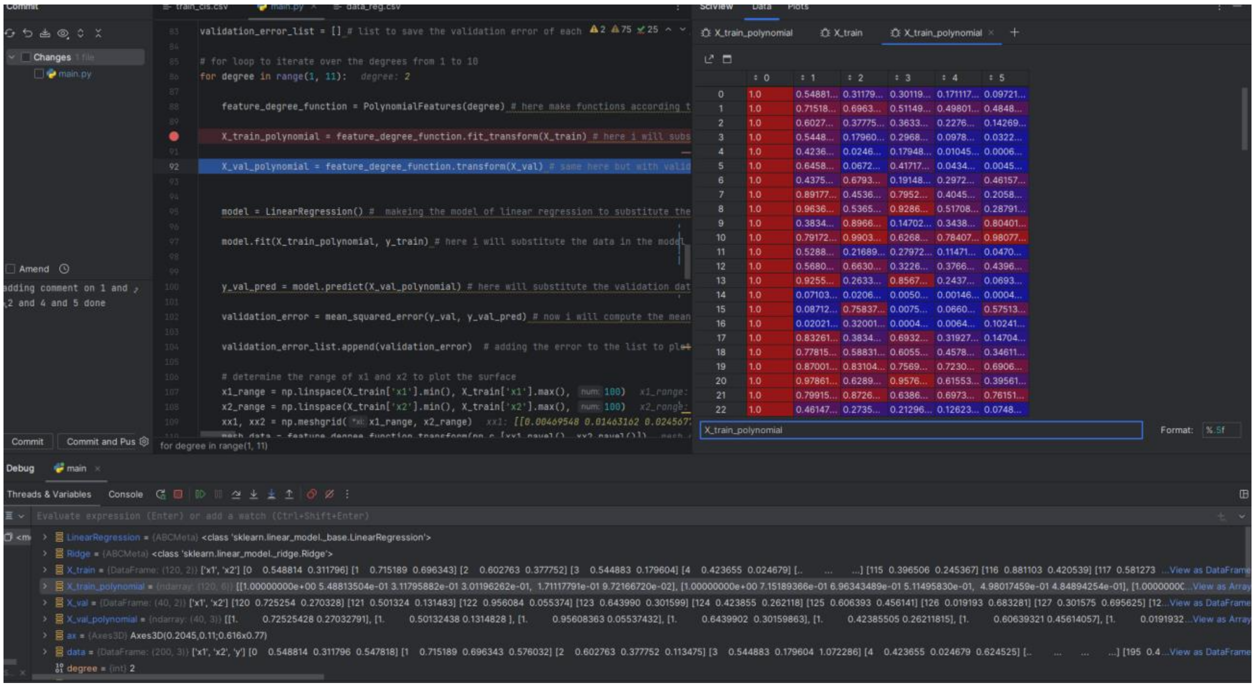


*Figure 9: part 2 features degree 2 displayed*

As we see here the array on the right shows the features of train data after substitute them, by poly function. and finally, we just substitute validation set using transform function.

We used **fit** transform in training data because we want it at first understand the structure of data and the degree of poly , then we don't need that on validation because it's already done above so we used transform alone.

Then by using **linearRegression ()** and model fit we found the module according to the degree assigned to it , then we make the prediction values by **model.predict(value_of_features_according to the dgree)**

Then we found the mean square error between the real values and predictions by mean function , then store that value in **val_errors** list to use it later to find the best degree for predictions , and plot figure showing that.

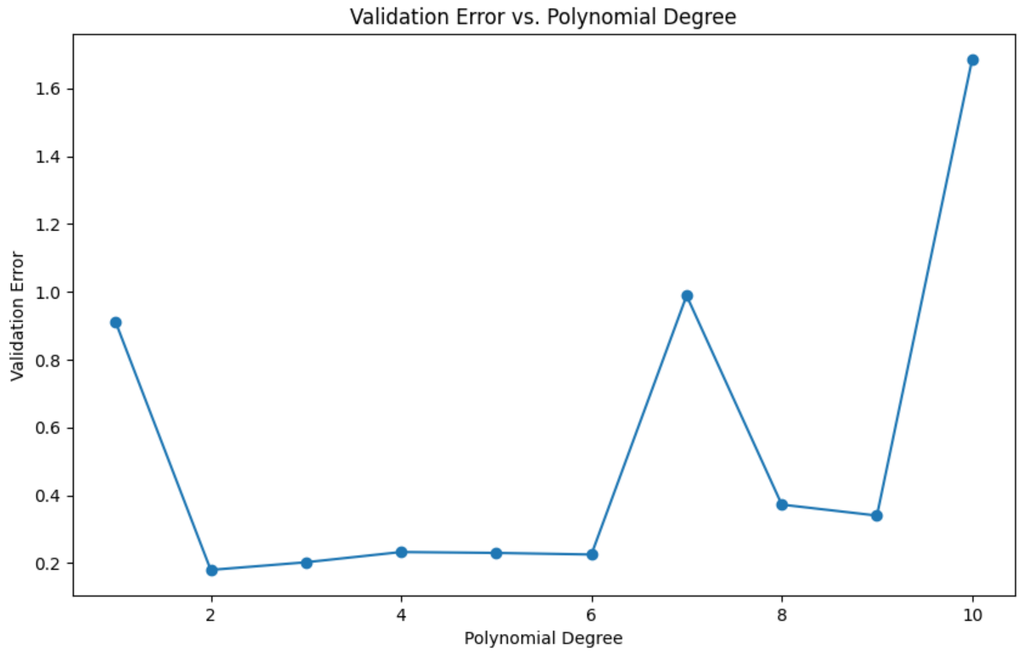



*Figure 10: Validation Error vs Polynomial Degree*

```
y_val_pred = model.predict(X_val_polynomial) # here will substitute the validation data in the model to predict the target feature.

validation_error = mean_squared_error(y_val, y_val_pred) # now i will compute the mean squared error of the validation data.

validation_error_list.append(validation_error)  # adding the error to the list to plot it later.
```

*Figure 11: how we get the validation error*

```
validation_error_list = {list: 10} [0.911824583849479, 0.17990349576391818, 0.2024332725287566, 0.23268739290547966, 0.22998004318855386, 0.22539808216993928, 0.9890112652181108, 0.3725126622710879, 0.34018777376608433, 1.6855103035270538]
    00 = {float64: ()} 0.911824583849479
    01 = {float64: ()} 0.17990349576391818
    02 = {float64: ()} 0.2024332725287566
    03 = {float64: ()} 0.23268739290547966
    04 = {float64: ()} 0.22998004318855386
    05 = {float64: ()} 0.22539808216993928
    06 = {float64: ()} 0.9890112652181108
    07 = {float64: ()} 0.3725126622710879
    08 = {float64: ()} 0.34018777376608433
    09 = {float64: ()} 1.6855103035270538
    __len__ = {int} 10
```

*Figure 12: validation error for each degree*

As we see, when we found the validation error by using mean square error between the predicted values and validation true values of y, we found that the least mean square error when the degree of the regression is 2.

And here is the surface plot for each degree from 1 to 10.

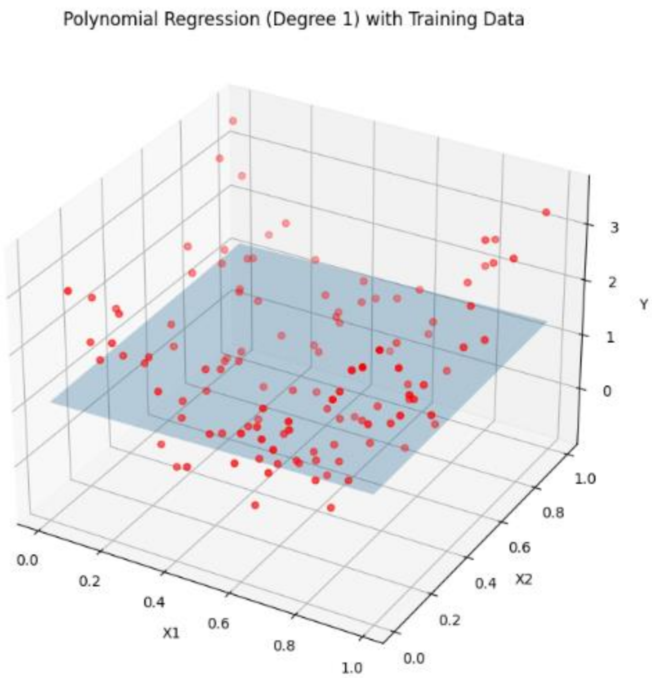



*Figure 13: degree 1 plot regression*

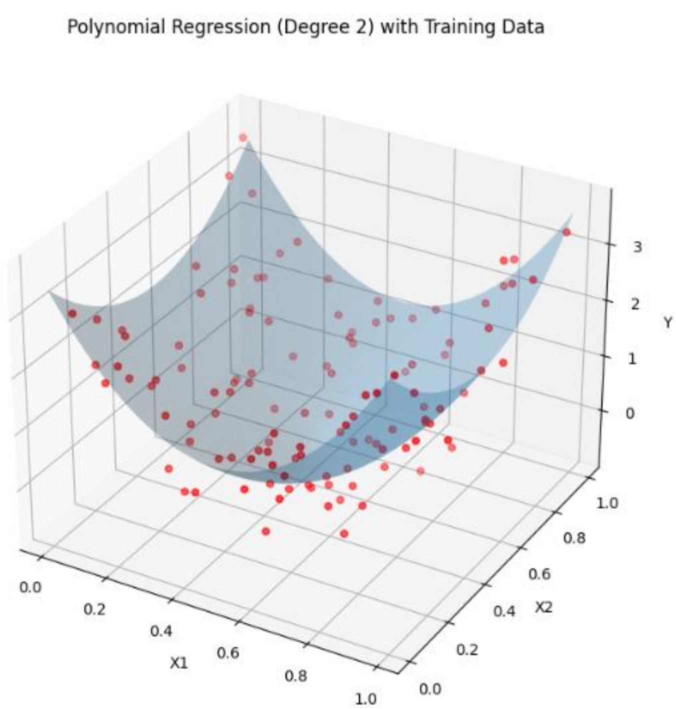



*Figure 14: degree 2*

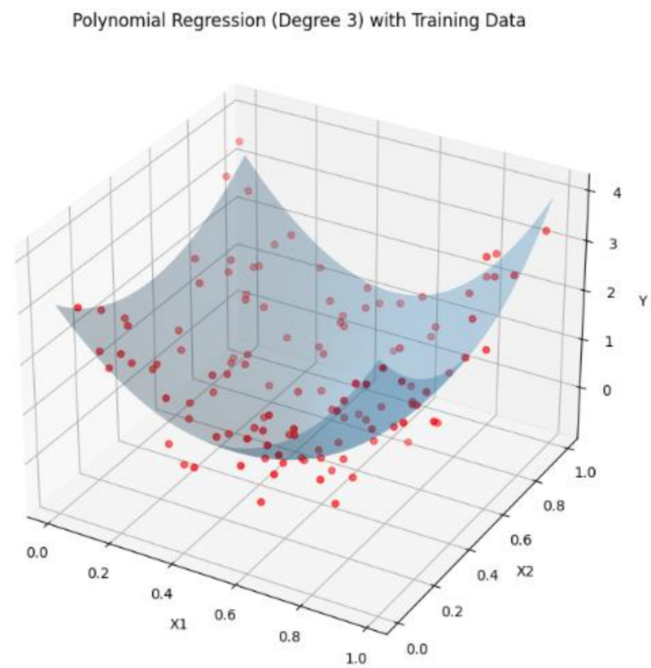



*Figure 15: degree 3*

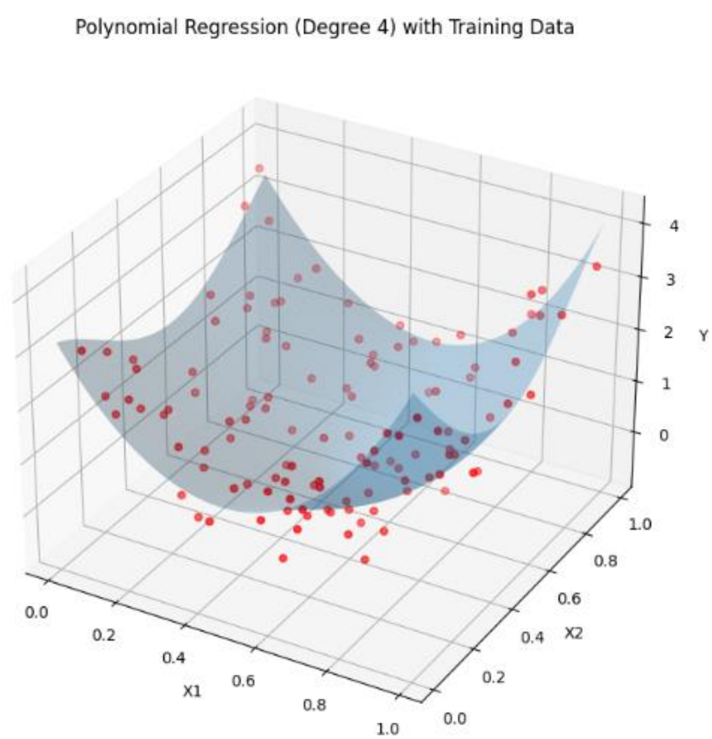



*Figure 16: degree 4*

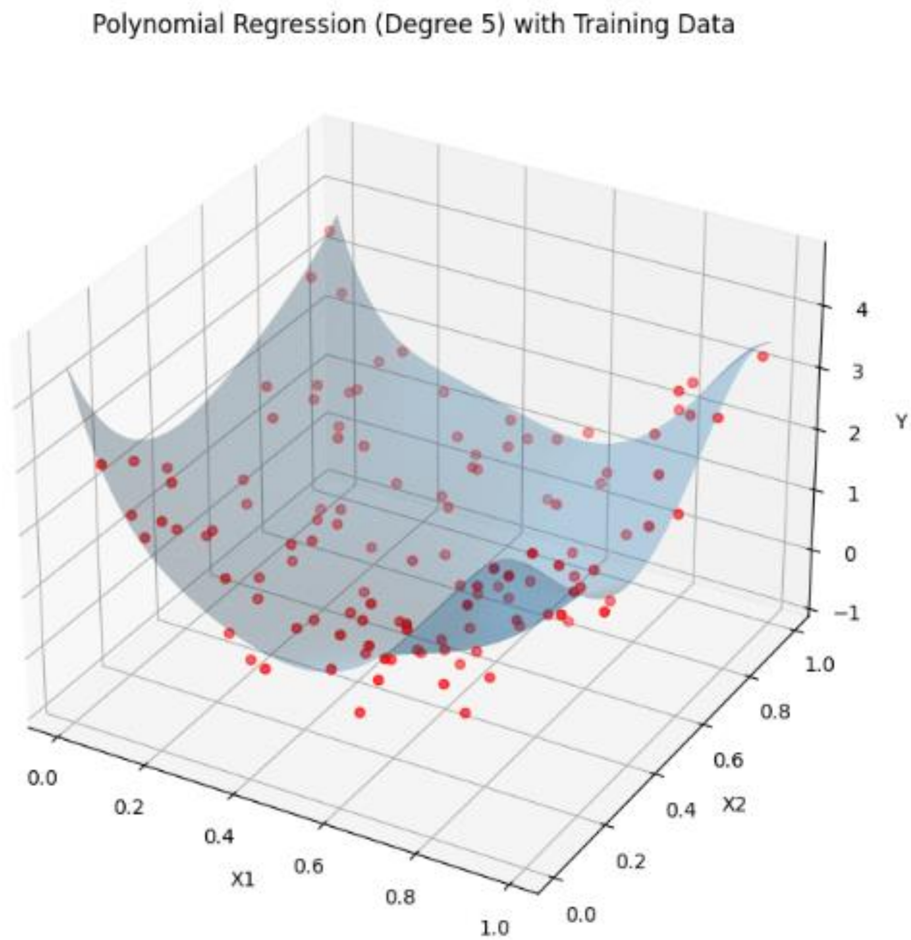



*Figure 17: degree 5*

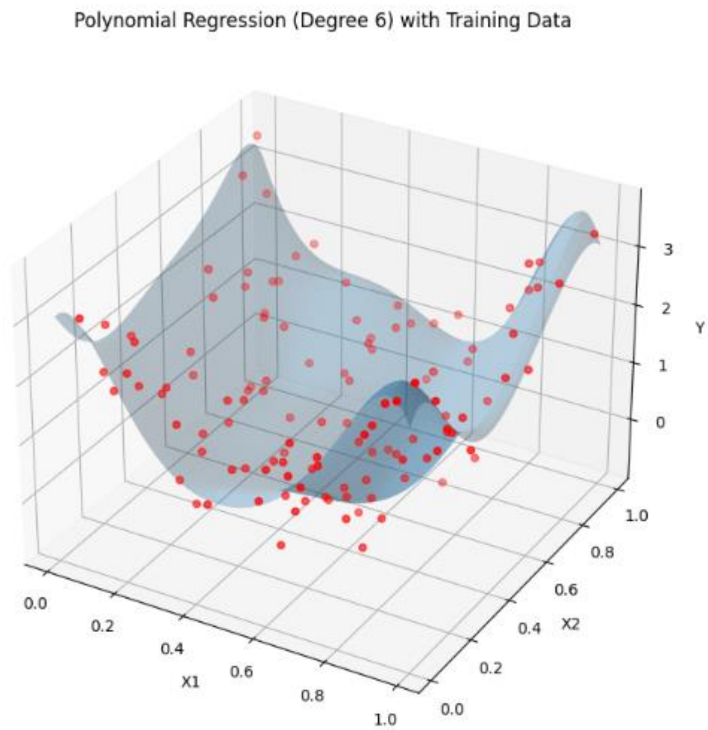



*Figure 18: degree 6*

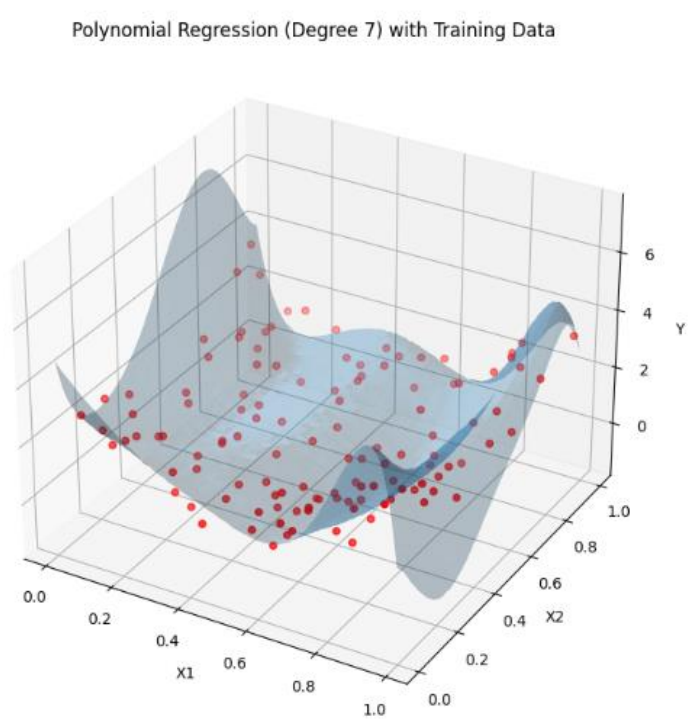



*Figure 19: degree 7*

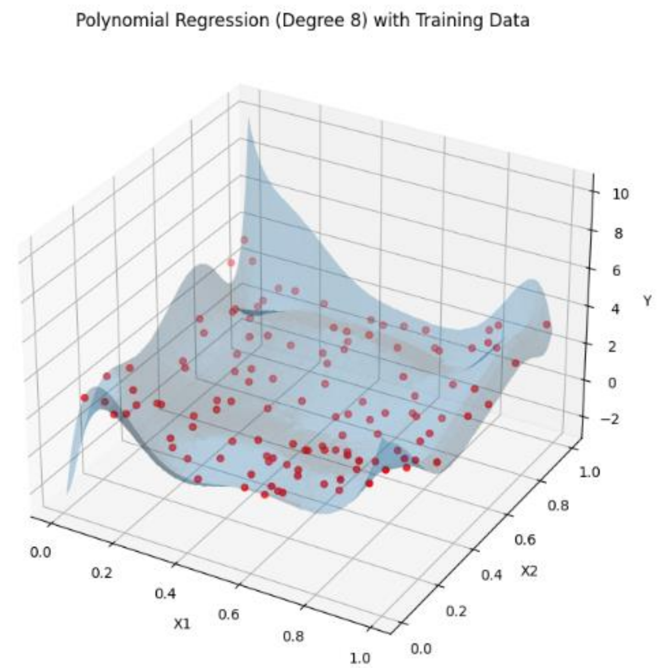



*Figure 20:degree 8*

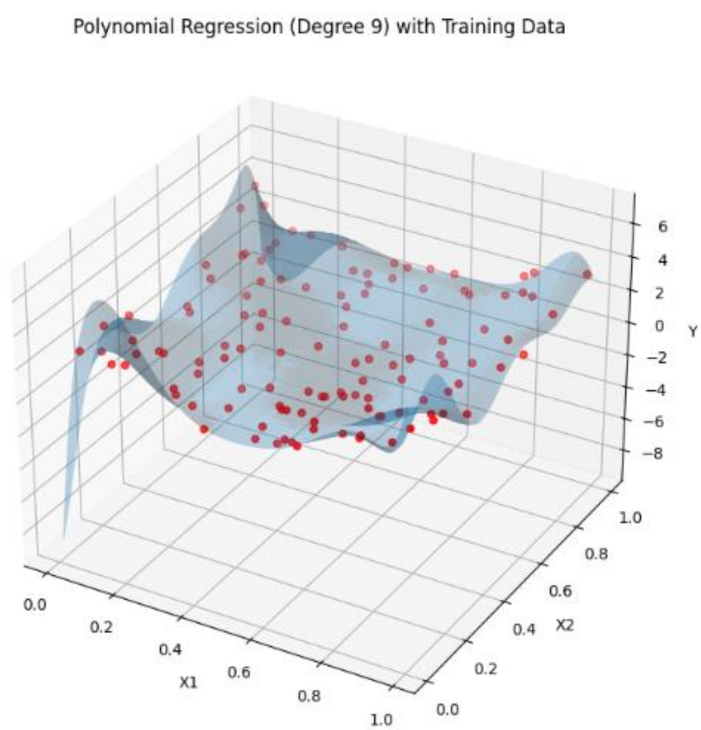



*Figure 21: degree 9*

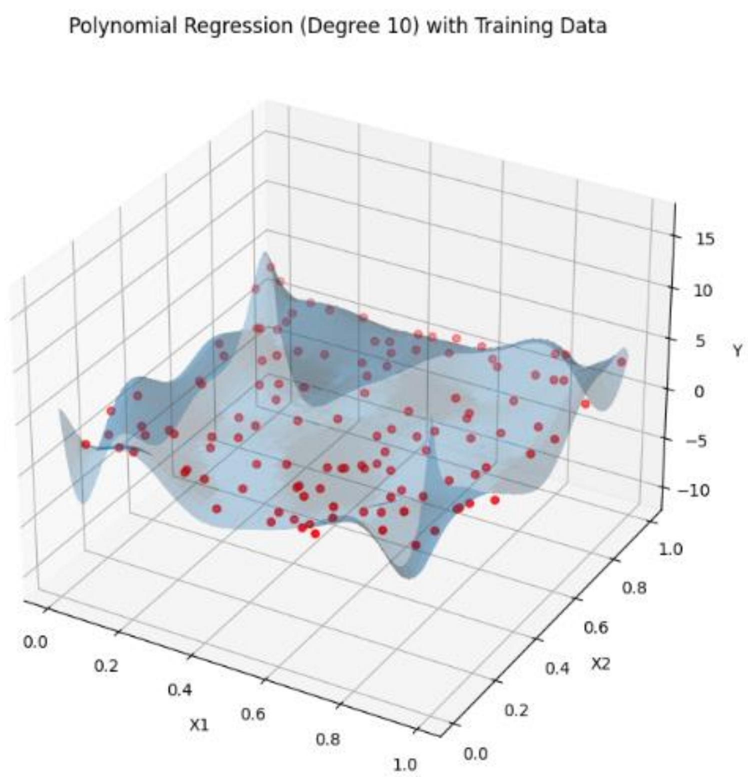



*Figure 22: degree 10*

## Part 3:

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`)

*Figure 23: part 3 text*

# Ridge Regression

- Alternatively, we can choose a regularization term that penalizes the squares of the parameter magnitudes. Then, our regularized loss function is:

$$L_{Ridge}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(\, y_i \; - \; \boldsymbol{w}^T\boldsymbol{x}_i \,)^2 + \lambda \sum_{j=1}^{d} w_j^2$$

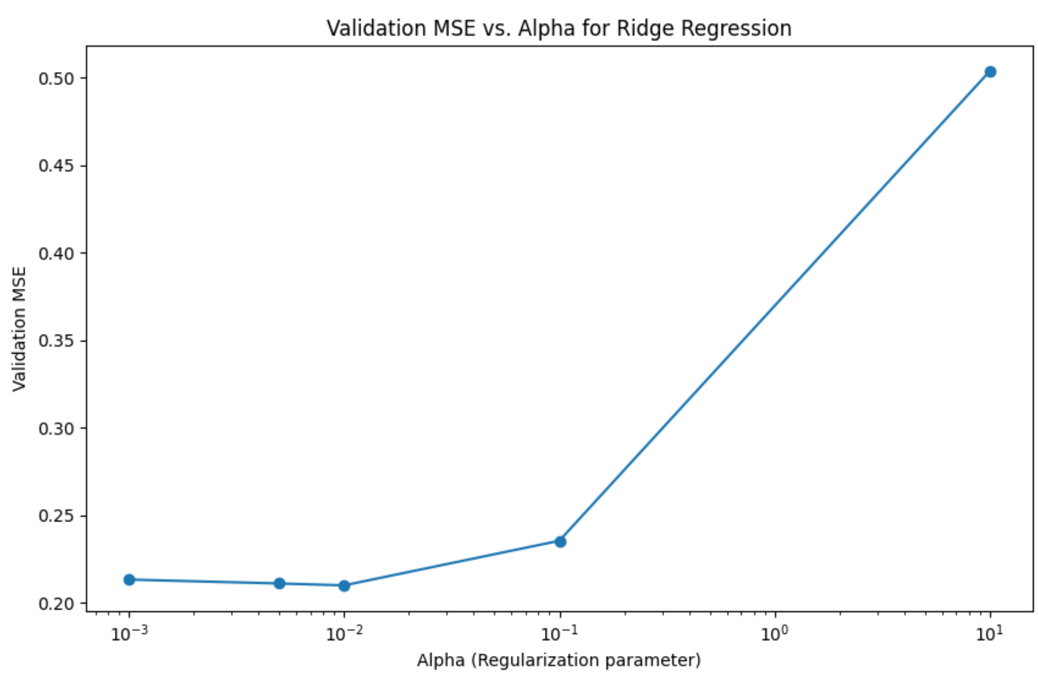



*Figure 24: validation error vs alpha values with Ridge regression*

```
alphas = {list: 5} [0.001, 0.005, 0.01, 0.1, 10]
    0 = {float} 0.001
    1 = {float} 0.005
    2 = {float} 0.01
    3 = {float} 0.1
    4 = {int} 10
    __len__ = {int} 5
```

*Figure 25: the values of alpahs*

```
val_errors = {list: 5} [0.21328335916735383, 0.21103328202646282, 0.20996554038850385, 0.23545304328552435, 0.5038254258404573]
    0 = {float64: ()} 0.21328335916735383
    1 = {float64: ()} 0.21103328202646282
    2 = {float64: ()} 0.20996554038850385
    3 = {float64: ()} 0.23545304328552435
    4 = {float64: ()} 0.5038254258404573
    __len__ = {int} 5
```

*Figure 26: the values of validation error according to each alpha using mean error with Ridge module.*

As we see here the best alpha that gave us the minimum error is alpha = 0.01 with error 0.2099 according to mean square error between the predicted values on the validation set on the ridge module and real values of output on the validation set y.

```
for alpha in alphas:  alpha: 10

    model = Ridge(alpha=alpha) # creating the model of ridge regression with the alpha value.

    model.fit(X_train_poly, y_train) # here i will substitute the data in the model to train it.


    y_val_pred = model.predict(X_val_poly) # getting the predicted values of the validation data.

    val_error = mean_squared_error(y_val, y_val_pred) # computing the mean squared error of the validation data.   val_error: 0.5038254258404573

    val_errors.append(val_error) # adding the error to the list to plot it later.
```

*Figure 27: how we get the validation error for each alpha*

First, we make the module using Ridge function given it the alpha in one iteration, then substitute the train set inside the function go get the final module when it trained by data. Then finding the error for each alpha by finding the mean error between the predicted validation set values and real values.

## Part 4

### 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.

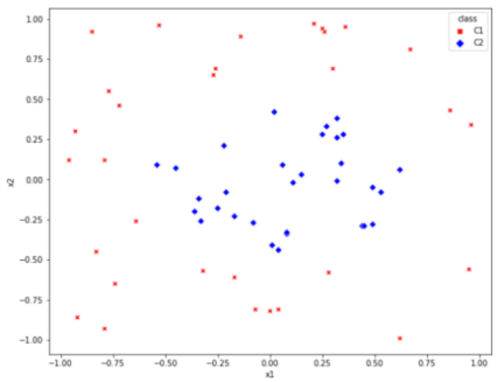



*Figure 1 Training Set*

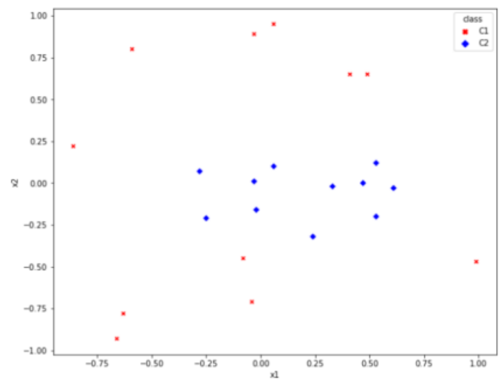



*Figure 2 Testing Set*

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 28: part 4 text*

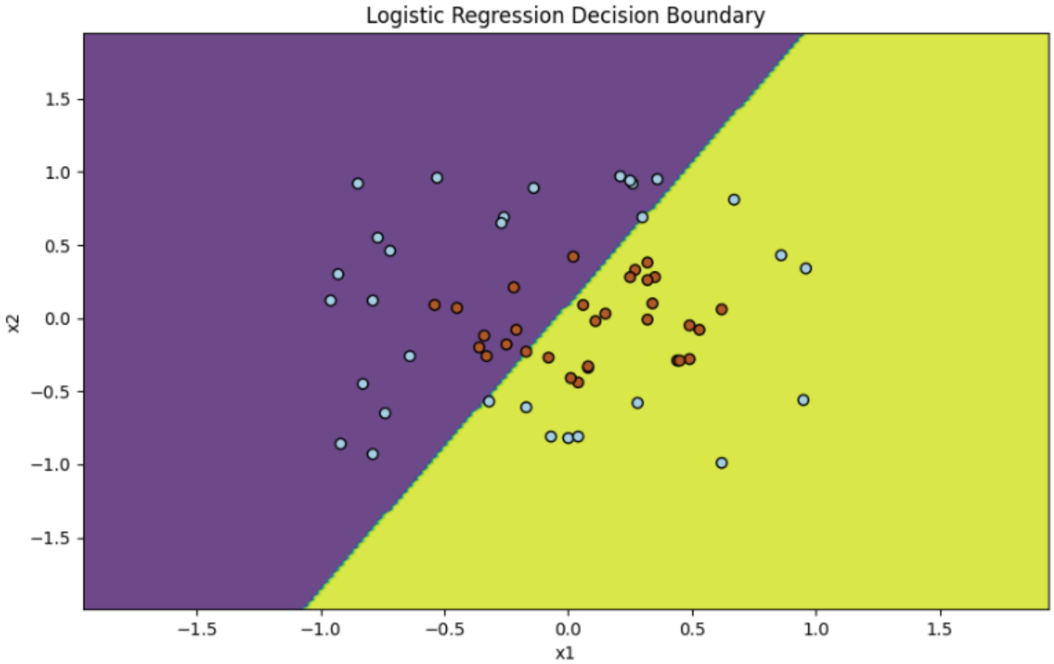



*Figure 29: Logistic regression part 4 Decision Boundary*

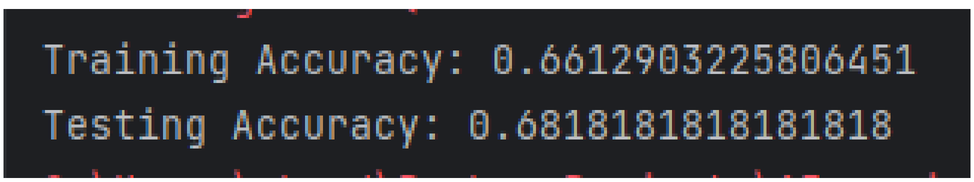
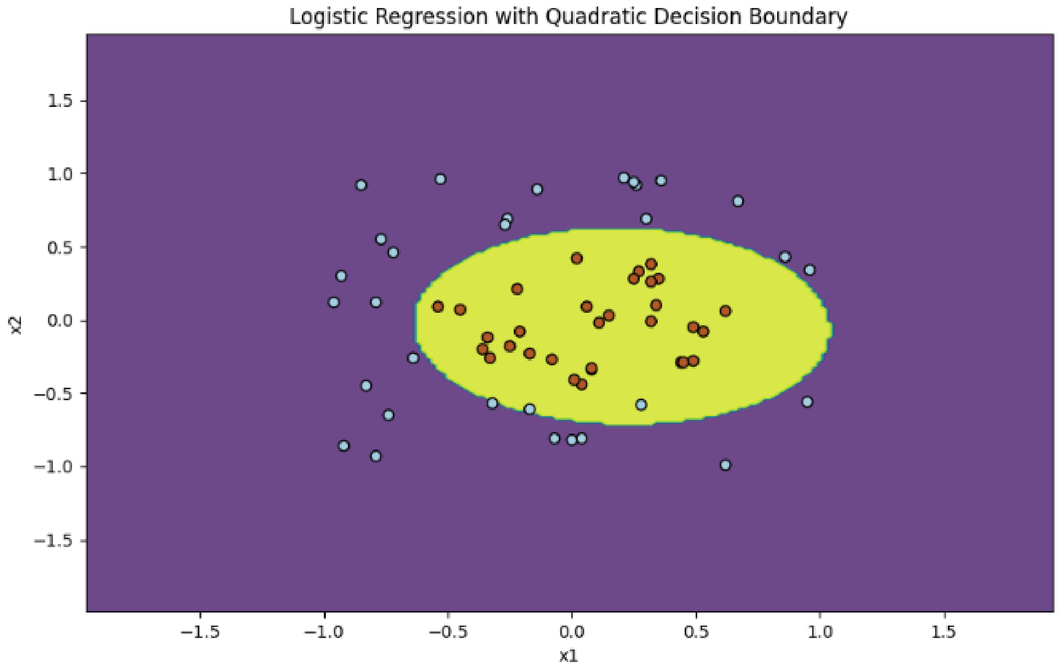



*Figure 30: accuracies of the module*

According to training accuracy and testing accuracy , this module appears **underfitting** in both of them , in order to low values that we got. I think the problem that the module that we create is too simple to get higher accuracy, and we need to make it more complex than liner , to better degree which gives us better results.

## Part 5

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




```
training accuracy of quadratic: 0.967741935483871
testing accuracy of quadratic: 0.9545454545454546
```

Now according to the both of training accuracy and testing, we see a big difference here in the results, this model gave us a higher accuracy in both training and testing, compared to linear one above, and in terms of overfitting this model not showing overfitting **due to the testing accuracy** which is high , and it more stable than the above one with linear model.

## Part 6

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

*Figure 31: part 6 text*

Answered above.