1B) Polynomial Regression and Regularization:

1.Give a brief description of your model, algorithms and how you implemented the regularization

Our model takes two parameters as inputs given in the question, x1 corresponding to the MLOGP and x2 to GATS1i.
Now since we have to plot the model for all degrees between 0 to 9 inclusively,
Our function is of the type:
Y = w0 + w1x1 + w2x2 for **degree one.**
Y= w0 + w0 + w1x1 + w2x2 + w3(x1^2) + w4(x1*x2) + w5(x2^2) for **degree two**
And so on…

Thus, for degree 'i', we will have (i+1)*(i+2)/2 number of terms including all the necessary powers.
Hence, degree 1 will have only 1 term (Bias) and degree 9 has a total of 55 terms including the bias term.

Since our model can contain outliers that can affect the model severely, we have Normalised the dataset for the **Training** data after applying an 80-20 split on the data.

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

This ensures all the training dataset values lie between 0 and 1.
This is similar to the fit-transform function of sklearn where we fit the training data and apply normalization transformation.

Important: The testing data is now transformed with the training data's normalized min and max values for every parameter.

Now we apply gradient descent on the normalized data using ita = 0.04, running over 100000 iterations.
This should give us the values of w which will result in the minimum error.
This is done over all the degrees ranging from 0 to 9.

$$L(x,y) = \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

where $h_\theta x_i = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$

$$L(x,y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

1.1

Figure 1.1 shows the error functions in use.
The above function is the **Unregularized** error (sum of squares)
The lower error function is the **Regularized** error.

Now, the gradient descent model runs over all the data points, over 100000 iterations. Thus, the value of w1 and w2 are changed after every iteration by using the below formula.

w(k+1) = w(k) - eta * (1/N) * dE/dw

This eventually results in the final w values for our training dataset.
Using the same values, we calculate errors for our testing dataset (the remaining 20% of the data).
Then we plot training and testing errors on a single plot and also the RMS(Root Mean Square) error.

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

The same process is done for the **Stochastic Gradient Descent Model**.
This is computationally less complex since we do randomly pick up one point from the dataset and use it to update the w values.

Given below is the code for the stochastic gradient descent model.
Alpha being our ita.
m is the number of data points.
Here, if the summation part is removed, it will result in Stochastic Gradient Descent.

Gradients

$$\theta_0 := \theta_0 - \alpha.(1/m.\sum_{i=1}^{m}(h(\theta^{(i)} - y^{(i)}).X_0^{(i)})$$

$$\theta_1 := \theta_1 - \alpha.(1/m.\sum_{i=1}^{m}(h(\theta^{(i)} - y^{(i)}).X_1^{(i)})$$

$$\theta_2 := \theta_2 - \alpha.(1/m.\sum_{i=1}^{m}(h(\theta^{(i)} - y^{(i)}).X_2^{(i)})$$

$$\theta_j := \theta_j - \alpha.(1/m.\sum_{i=1}^{m}(h(\theta^{(i)} - y^{(i)}).X_0^{(i)})$$

Given below is our implementation for the same process for the **Vector Gradient Descent Model**.

```
model_arr = []
cumulative_w_for_degrees = []
degree_errors = []
#Ranging for all polynomials from degree 0 to degree 9
for k in range(0,10):
    #Extracting total number of parameters in the regression model
    no_of_coeffients = ((k+1)*(k+2))//2
    #Selecting required variables as per the given regression model
    X_selected_degree = X_new[:,:no_of_coeffients]
    X_selected_degree_transpose = X_selected_degree.T
    w_for_selected_degree = w[:no_of_coeffients,:]
    for i in range(iterations_gradient_descent):
        #Calculate predicted value of y
        calcuated_y = X_selected_degree.dot(w_for_selected_degree)
        #Differentiating the error term with respect to the parameters
        diff = X_selected_degree_transpose.dot(calcuated_y-y_actual_train)
        #using gradient descent method to update the w's
        w_for_selected_degree = w_for_selected_degree -(1/training_datasize) * ita * diff

    model = X_selected_degree.dot(w_for_selected_degree)
    #Appending the final y-predicted values in the model
    model_arr.append(model)
    cumulative_w_for_degrees.append(w_for_selected_degree)
    training_errors = []
    #Appending the error generated by the model for every degree from 0-9
    degree_errors.append(calculateError(X_selected_degree, w_for_selected_degree))

for items in degree_errors:
    for item in items:
        training_errors.append(item[0])
rmse_train = calculateError2(training_errors)
for i in cumulative_w_for_degrees:
    print(i)
```

Regularisation:

We have applied regularisation in our model as well. This restricts the growth of w's and ensures overfitting is reduced.
As per the given question, regularisation on q values [0.5,1,2,4] over several lambda values.
We also included lambda = 0 to show model remains unregularized for lambda = 0.
We also used a high lambda of value to show that w1 and w2 tend to zero in that case since they are highly restricted. This model effectively becomes a 0-degree polynomial.

Implementation:

```python
#Regularizing the parameters i.e. restricting the growth of parameters
def penalty_regularisation(w,q,iterations_,l):
    vec = np.zeros((3,1), dtype=float)

    if iterations_!=0:
        for i in range(1,3):
          #Adding the parameter regularzed term to the error function
            vec[i] = 0.5*(l * q * w[i][0] * (abs(w[i][0])**(q-2)))

    return vec
```

0.5s

```python
no_of_coeffients = 3
iterations_for_regularised = 100000
itar = 0.004
given_arr = [0.5,1,2,4]
lambda_array = [10000,1,0.1,0.25,0.75,0]
for lambda_counter in range(len(lambda_array)):
    for k in range(len(given_arr)):
        q = given_arr[k]
        X_regul = single_df.to_numpy()
        X_T_regul = X_regul.T
        w_regularised = w[:no_of_coeffients,:]
        for i in range(iterations_for_regularised):
            h_t_reg = X_regul.dot(w_regularised)
            #Unregularized error term
            sum_squared_errors = X_T_regul.dot(h_t_reg-y_actual_train)
            #Adding the regularized term to sum_squared_errors
            total_regularised_error = sum_squared_errors + penalty_regularisation(w_regularised,q,i,lambda_array[lambda_counter])
            #Applying Gradient Descent Method
            w_regularised = w_regularised -(1/training_datasize) * itar * total_regularised_error
        #Appending the parameters to the regularized-model
        print(f"w values for lambda = {lambda_array[lambda_counter]} and q = {q}: ")
        print(w_regularised)
        print("\n")
```

2.Tabulate the training and testing errors obtained using polynomial regression models of various degrees and your observations on overfitting.

Given below are the tabulated RMS training and testing errors for both stochastic as well as vector gradient descent method errors.

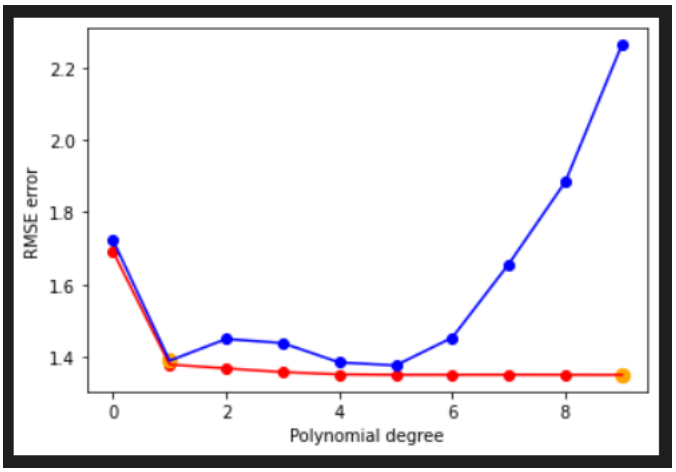
	rmse_train	rmse_test	rmse_train_stoc	rmse_test_stoc
0	1.690783	1.722103	1.691521	1.712294
1	1.378350	1.388873	1.378777	1.380137
2	1.367732	1.449075	1.368197	1.462887
3	1.357432	1.437760	1.367686	1.394513
4	1.351264	1.384355	1.351971	1.388918
5	1.349886	1.375867	1.350145	1.370823
6	1.349981	1.452808	1.350567	1.481869
7	1.350215	1.656291	1.351490	1.619393
8	1.349912	1.883509	1.354197	1.905742
9	1.349643	2.262312	1.401707	2.251513

We can also see the same trend in total (sum of the square of errors) below for all 4 possibilities above.
NOTE: The sum of squares of testing errors is less since the number of data points is 4 times less too.

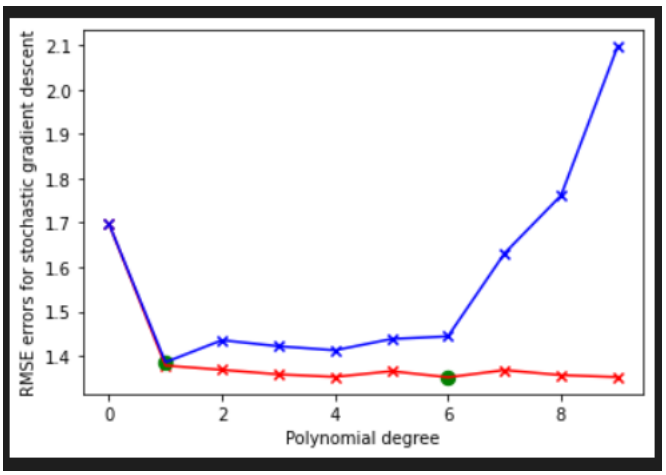
	total_train	total_test	total_train_stoc	total_test_stoc
0	1249.272559	263.941938	1250.659765	267.975639
1	830.233473	171.678210	830.949509	174.970486
2	817.492101	186.883723	841.775189	214.068359
3	805.225666	183.976683	810.239402	195.593667
4	797.924478	170.563131	799.040151	169.079124
5	796.298282	168.477769	814.346416	182.542547
6	796.410528	187.847996	796.490445	191.454849
7	796.686590	244.153660	798.929586	248.393528
8	796.328719	315.737017	804.479417	310.659239
9	796.011728	455.506920	798.598320	489.800107

The plot of Polynomial degrees vs RMS error.
We can see that minimum Testing error occurs at degree 1.
So the best model for our case is the one-degree polynomial regression model.

As and when we increase the degree, the testing error increases exponentially.
This is due to overfitting.
The bias vs variance is self-indicative in the given plot.



Similarly for stochastic errors.
Note that results obtained using stochastic are less accurate although the computation is heavily decreased hence is a good tradeoff.



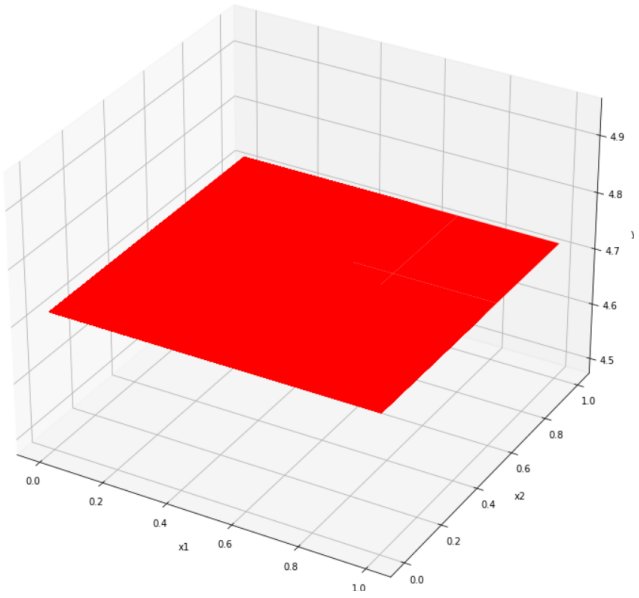
The training errors visibly decrease as and when the degree increases.
This is because our model tries to **overfit** the training dataset too much.

GREEN AND ORANGE DOT indicate the best-fitted model (Degree 1 model).

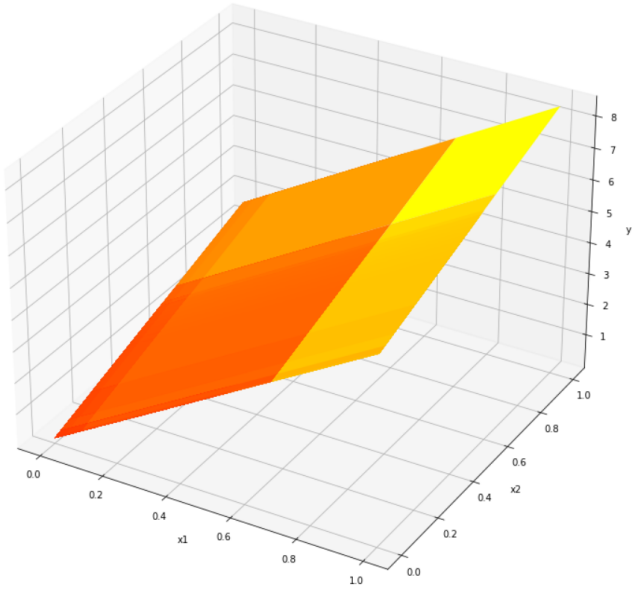
3.Surface plots of the predicted polynomials (Plot of $x1$, $x2$ vs y($x1$, $x2$) where y is the predicted polynomial.)

Surface Plots

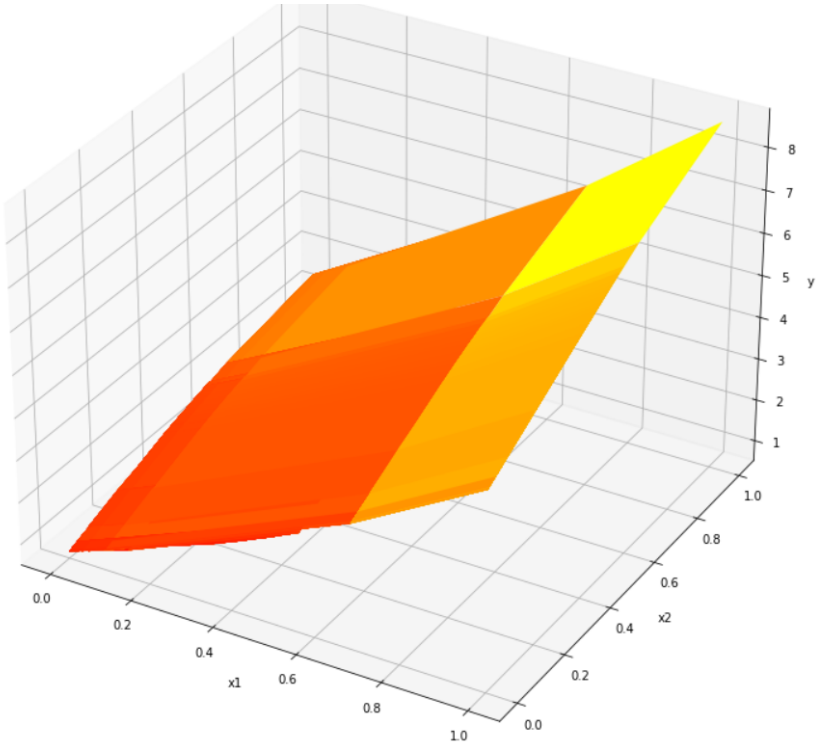
Degree 0:



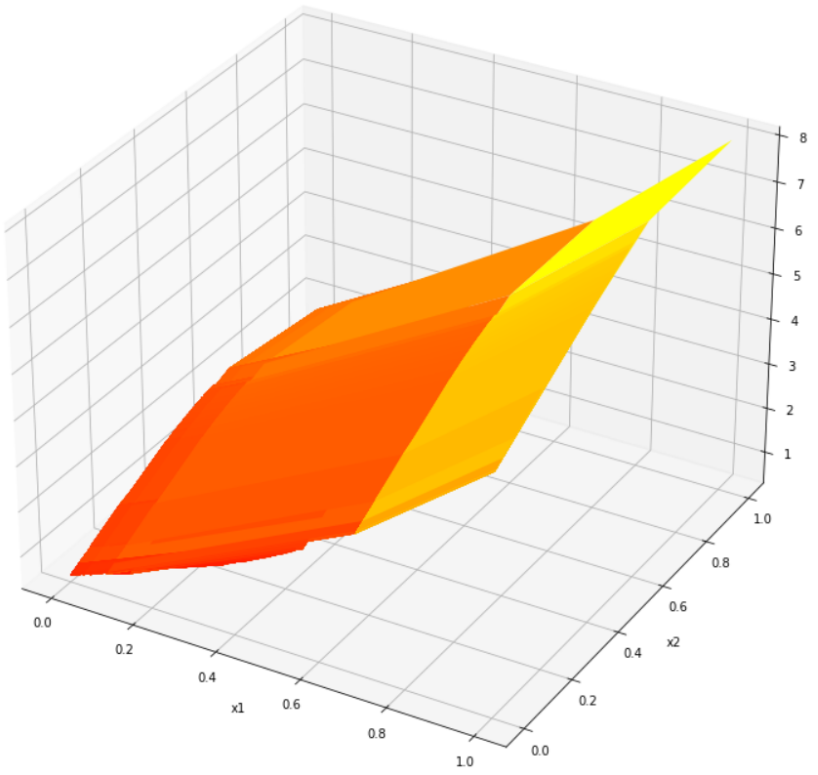
Degree 1:



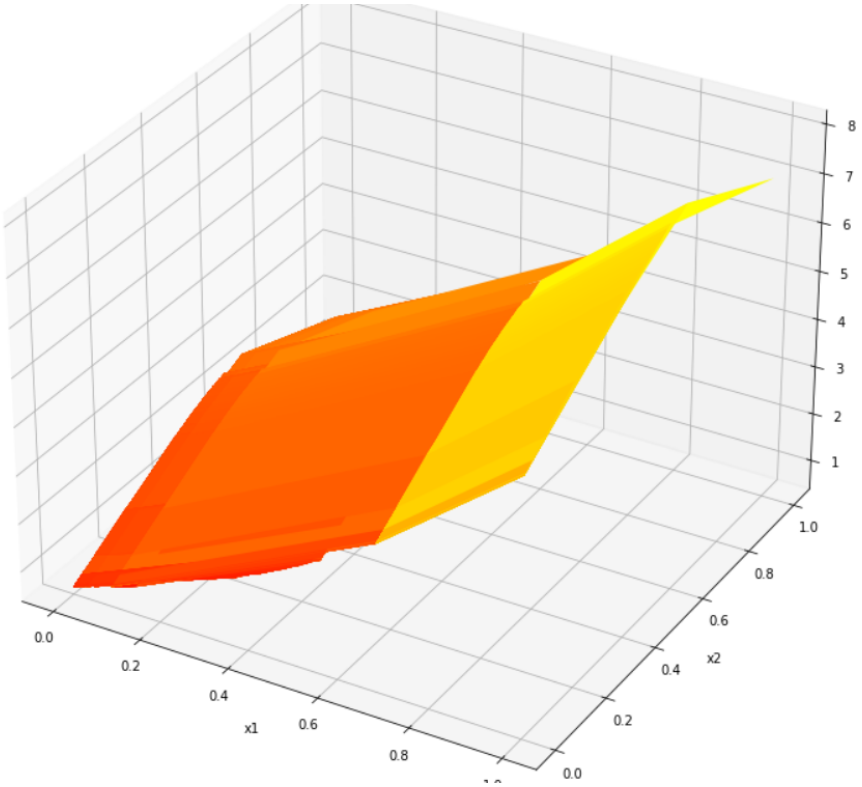
Degree 2:



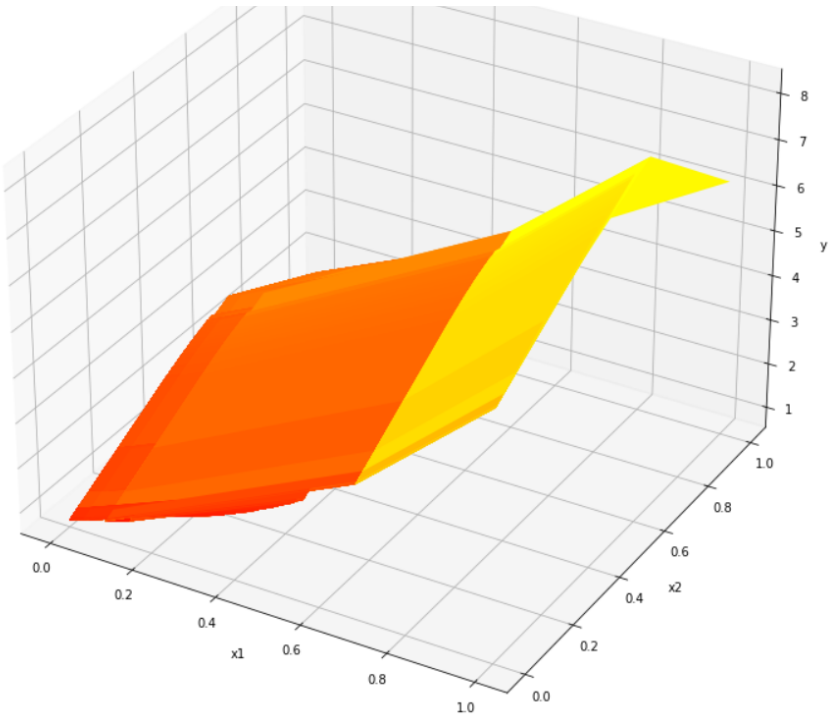
Degree 3:



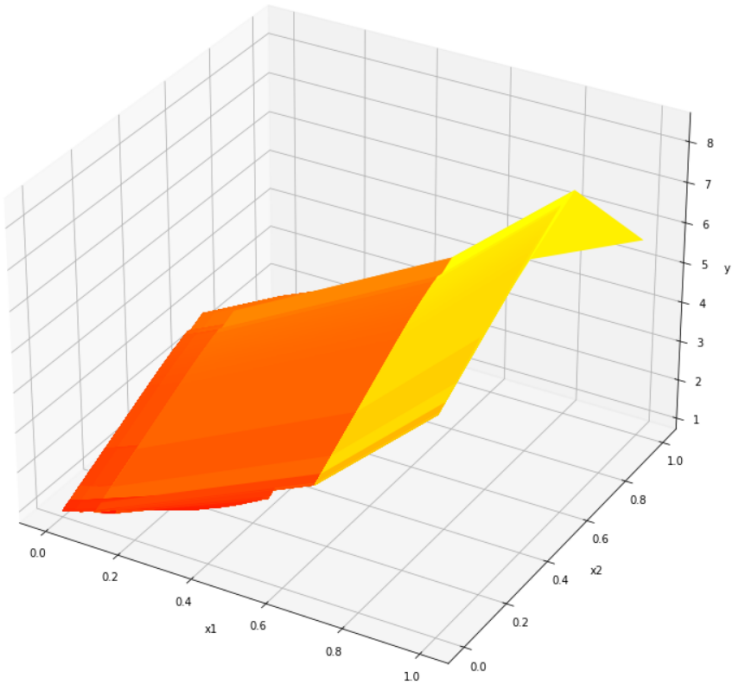
Degree 4:



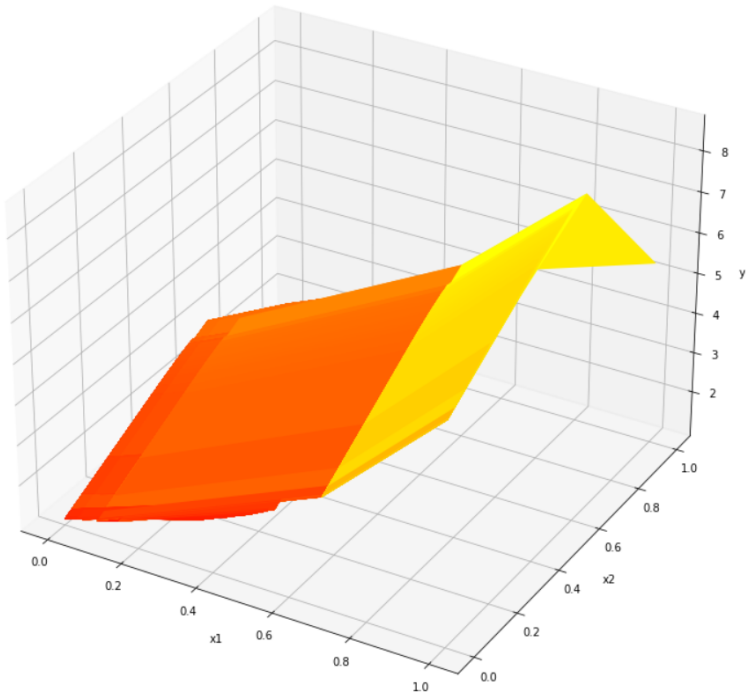
Degree 5:



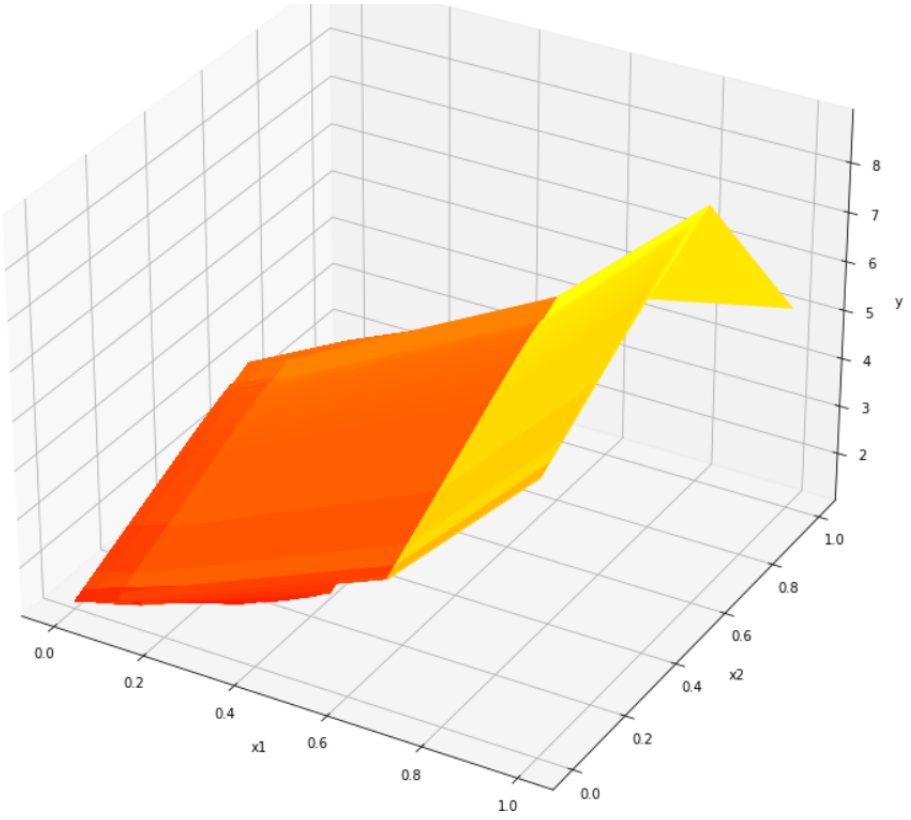
Degree 6:



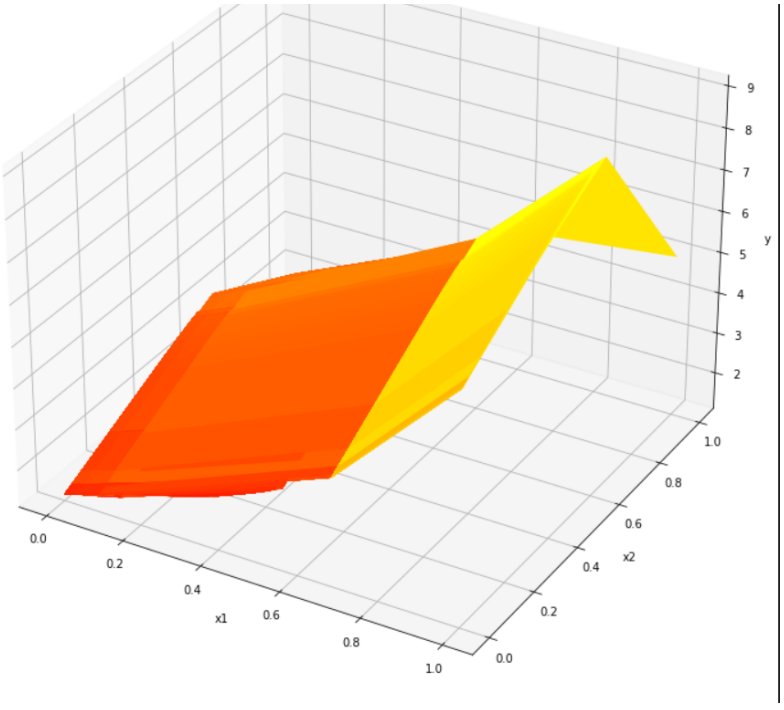
Degree 7:



Degree 8:



Degree 9:



4.The comparative analysis study of the four optimal regularized regression models and best-fit classic polynomial regression model:

Since the best fit occurs at degree one polynomial. Its Unregularized error is 171.6782. Comparing the given error values for every best model obtained for each value of q using some values of lambda we check which model gives us the minimum error. Thus the result obtained are:

(As shown in the snippets)

q = 2 (minimum error at lambda = 1)

```
w values for lambda = 10000 and q = 2:
[[4.71351226e+00]
 [5.12503296e-03]
 [4.66969286e-03]]


Error for given model:
[[263.69127074]]


w values for lambda = 10 and q = 2:
[[2.41196546]
 [2.63776756]
 [1.79372136]]


Error for given model:
[[185.72695504]]


w values for lambda = 1 and q = 2:
[[0.51752273]
 [5.15962369]
 [2.50323568]]


Error for given model:
[[170.42173948]]


w values for lambda = 0.1 and q = 2:
[[0.12270261]
 [5.7267825 ]
 [2.561936  ]]


Error for given model:
[[171.45873043]]


w values for lambda = 0.5 and q = 2:
[[0.30754214]
 [5.45959742]
 [2.53801253]]


Error for given model:
[[170.80851472]]


w values for lambda = 0 and q = 2:
[[0.07388739]
 [5.79782243]
 [2.56723296]]


Error for given model:
[[171.67821032]]
```

q = 1 (minimum error at lambda = 10)

```
w values for lambda = 10000 and q = 1:
[[ 4.71725431e+00]
 [-3.13368237e-02]
 [-1.75294033e-03]]


Error for given model:
[[263.38413846]]


w values for lambda = 10 and q = 1:
[[0.52821771]
 [5.24898317]
 [2.27665972]]


Error for given model:
[[169.77655722]]


w values for lambda = 1 and q = 1:
[[0.11932042]
 [5.7429385 ]
 [2.53817564]]


Error for given model:
[[171.37590624]]


w values for lambda = 0.1 and q = 1:
[[0.07843069]
 [5.79233403]
 [2.56432723]]


Error for given model:
[[171.64685853]]


w values for lambda = 0.5 and q = 1:
[[0.0966039 ]
 [5.77038046]
 [2.5527043 ]]


Error for given model:
[[171.52394331]]


w values for lambda = 0 and q = 1:
[[0.07388739]
 [5.79782243]
 [2.56723296]]


Error for given model:
[[171.67821032]]
```

q = 4 (minimum error at lambda = 0.1)

```
w values for lambda = 10000 and q = 4:
[[4.58782933]
 [0.13564387]
 [0.13086103]]


Error for given model:
[[257.25326405]]


w values for lambda = 10 and q = 4:
[[3.53827191]
 [1.24641458]
 [1.13985101]]


Error for given model:
[[213.52194284]]


w values for lambda = 1 and q = 4:
[[2.50592534]
 [2.3991714 ]
 [2.00277763]]


Error for given model:
[[187.04061783]]


w values for lambda = 0.1 and q = 4:
[[1.20772008]
 [4.05146671]
 [2.65190338]]


Error for given model:
[[172.77308671]]


w values for lambda = 0.5 and q = 4:
[[2.11831696]
 [2.86025828]
 [2.26595653]]


Error for given model:
[[180.87629106]]


w values for lambda = 0 and q = 4:
[[0.07388739]
 [5.79782243]
 [2.56723296]]
```

q = 0.5 (minimum error at lambda = 10)

```
w values for lambda = 10000 and q = 0.5:
[[-0.27500474]
 [ 0.04053277]
 [18.40000993]]


Error for given model:
[[995.00648731]]


w values for lambda = 10 and q = 0.5:
[[0.17396087]
 [5.69806439]
 [2.45781568]]


Error for given model:
[[170.87995241]]


w values for lambda = 1 and q = 0.5:
[[0.08380187]
 [5.78787009]
 [2.55654135]]


Error for given model:
[[171.59217184]]


w values for lambda = 0.1 and q = 0.5:
[[0.07487793]
 [5.79682746]
 [2.56616615]]


Error for given model:
[[171.66954611]]


w values for lambda = 0.5 and q = 0.5:
[[0.07884212]
 [5.792847  ]
 [2.56189371]]


Error for given model:
[[171.63502322]]


w values for lambda = 0 and q = 0.5:
[[0.07388739]
 [5.79782243]
 [2.56723296]]


Error for given model:
[[171.67821032]]


Error for best fit unregularised: (degree=1)
[[171.67821032]]
```