Description of the model:

With the help of provided code, we can produce 18 different types of models - 9 models ranging from degrees 0-9 formed using gradient descent, and nine models ranging from degrees 0-9 formed using the ***stochastic gradient descent method***.

Apart from this, we can also provide the optimum lambda value for the different regularization methods implemented. In addition, we can also plot the Surface plots of the predicted polynomials (Plot of x_1, x_2 vs. y(x_1, x_2) where y is the predicted polynomial.)

We can also modify the learning rate at which the model adapts as well as the number of iterations it runs for(if the system cannot handle many iterations)

We implemented the algorithms in the following way:

1. Initially, we read the data from the CSV file provided using the pandas module present in python. Then we normalize the data using appropriate methods.

2. Next, we perform an 80-20 train-test split for training the model and testing it later.

3. Next, we defined a function called **GradDesc**, which implements the basic principle behind gradient descent of updating the w values

Gradient Descent algorithm is an optimization algorithm used to estimate the coefficients that minimize the given cost function. It is an iterative algorithm since it applies the same w_j formula on the coefficients repeatedly until we reach the desired precision.

$$\mathbf{w = w_0 - eta * (gradient(X, w_0, y)),}$$

gradient function calculates the gradient of the error function at the point w_0.

Results Using gradient descent, learning rate =0.001

Degree	Training Loss	Testing Loss
0	6.111693694477511244	1.5622206617194221918
1	4.1504344074169947478	0.9299447145208079344
2	3.9963999487206197624	0.89167944687431437687
3	3.986126048055051729	0.89016862444933609857
4	3.970504663258521487	0.88879824533593414085
5	3.9580000329468465715	0.88841542068615563904
6	3.9513784097763294902	0.8890128595289065334
7	3.9483671490093141796	0.8897976006437402069
8	3.9469232362155018236	0.89038407316807991143
9	3.9460896482696125314	0.89072334291327607994

4. We have also defined a function called **Polynomial_features**, which computes all the permutations of X_1 and X_2 depending on the input degree. A column of 1 is also added in the feature matrix to accommodate for the constant w_0 in the regression equation.

Working of polynomial feature:

```
X = np.array(
    [[1,2,3],
     [1,5,6],
     [1,1,2],
     [1,9,2]])

Polynomial_features(X,2)  # degree = 2
```

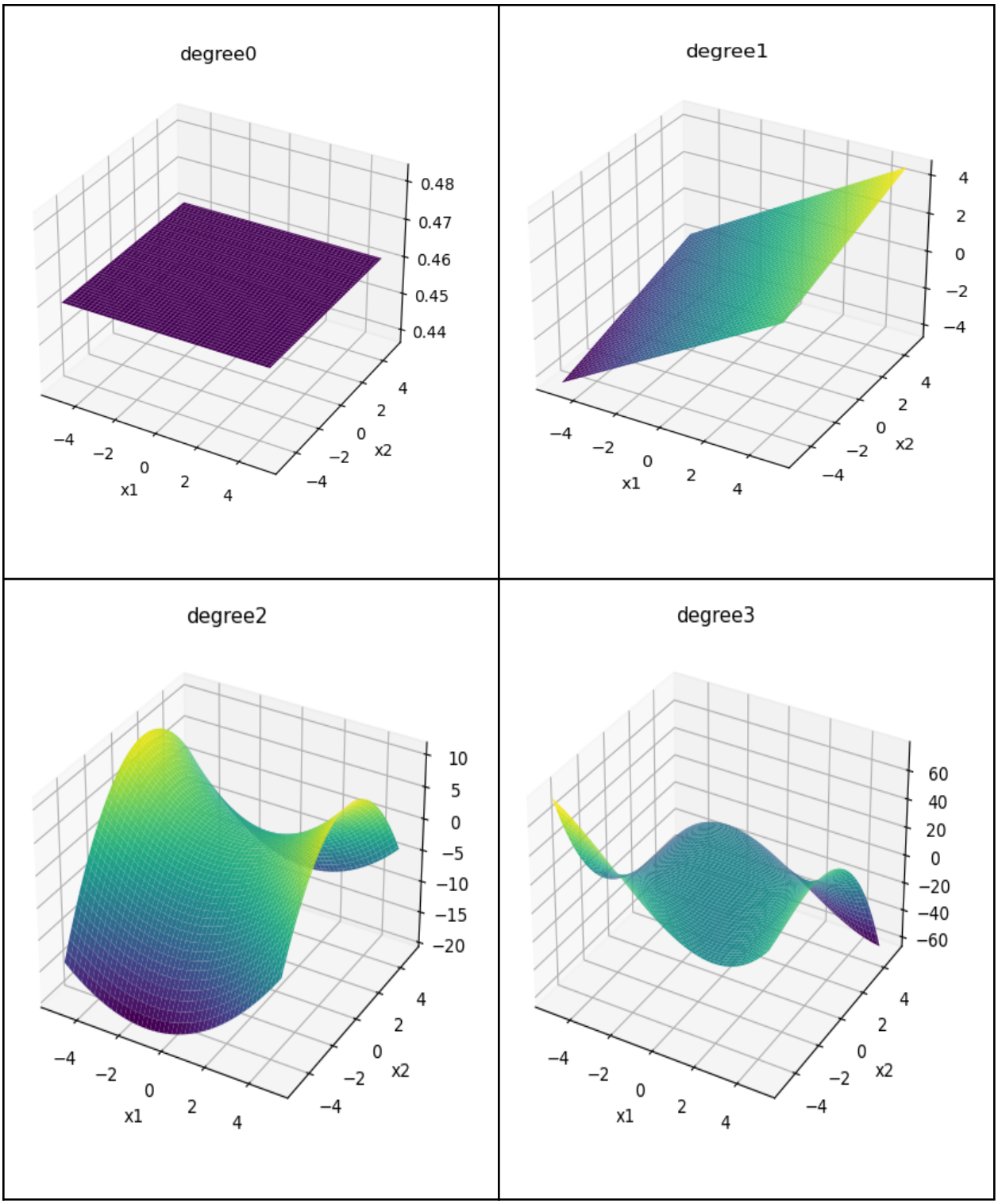
```
array([[ 1,  2,  4,  3,  6,  9],
       [ 1,  5, 25,  6, 30, 36],
       [ 1,  1,  1,  2,  2,  4],
       [ 1,  9, 81,  2, 18,  4]])
```

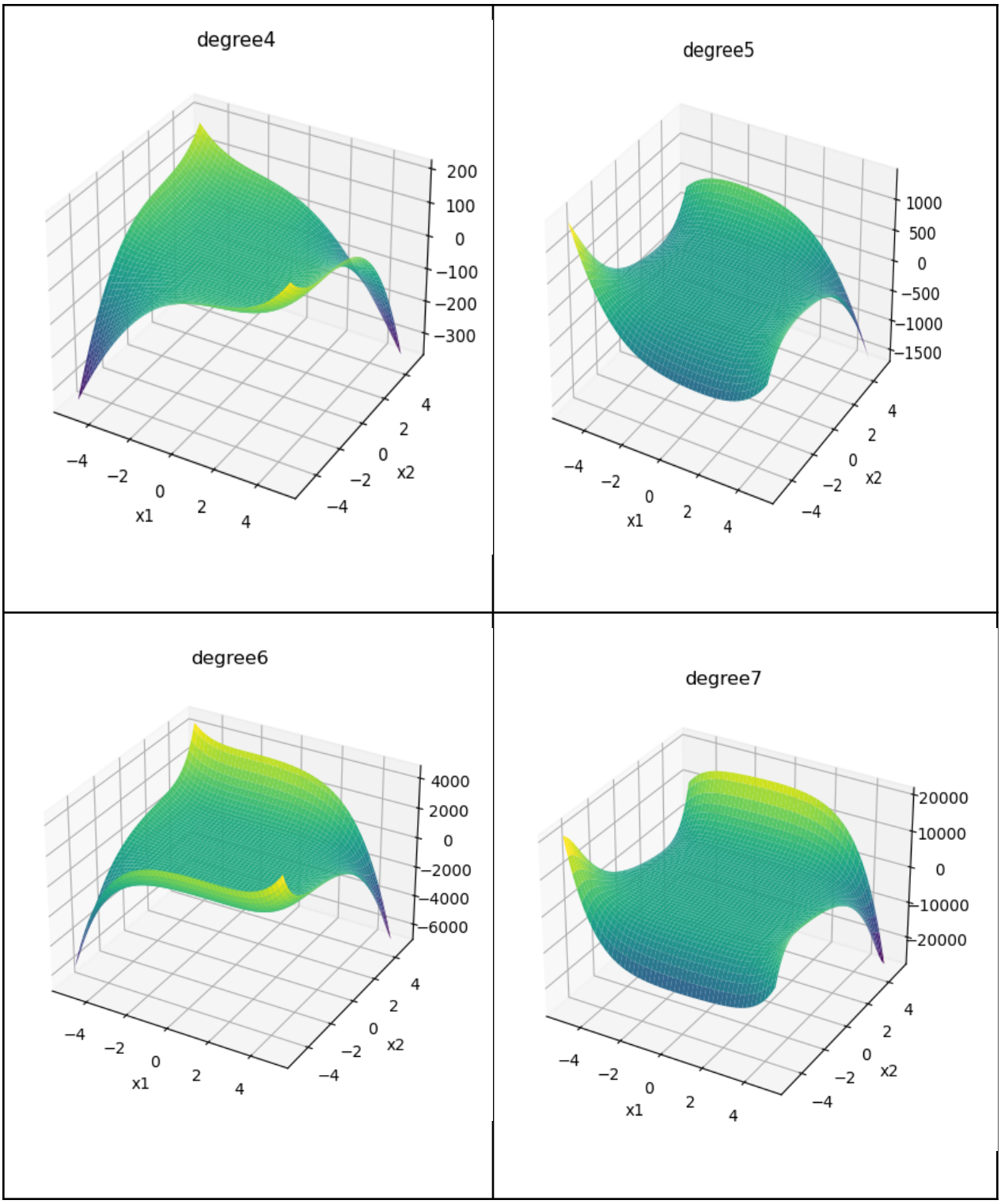
Stochastic Gradient Descent, a single sample is selected randomly instead of the whole data set for each iteration. That single data sample is then used to update the values of weight vector in each iteration. It has been observed that stochastic gradient descent also yields the same result as the gradient descent algorithm and, in some cases, gives a better result. Also, it is takes very less computations as compared to the gradient descent.

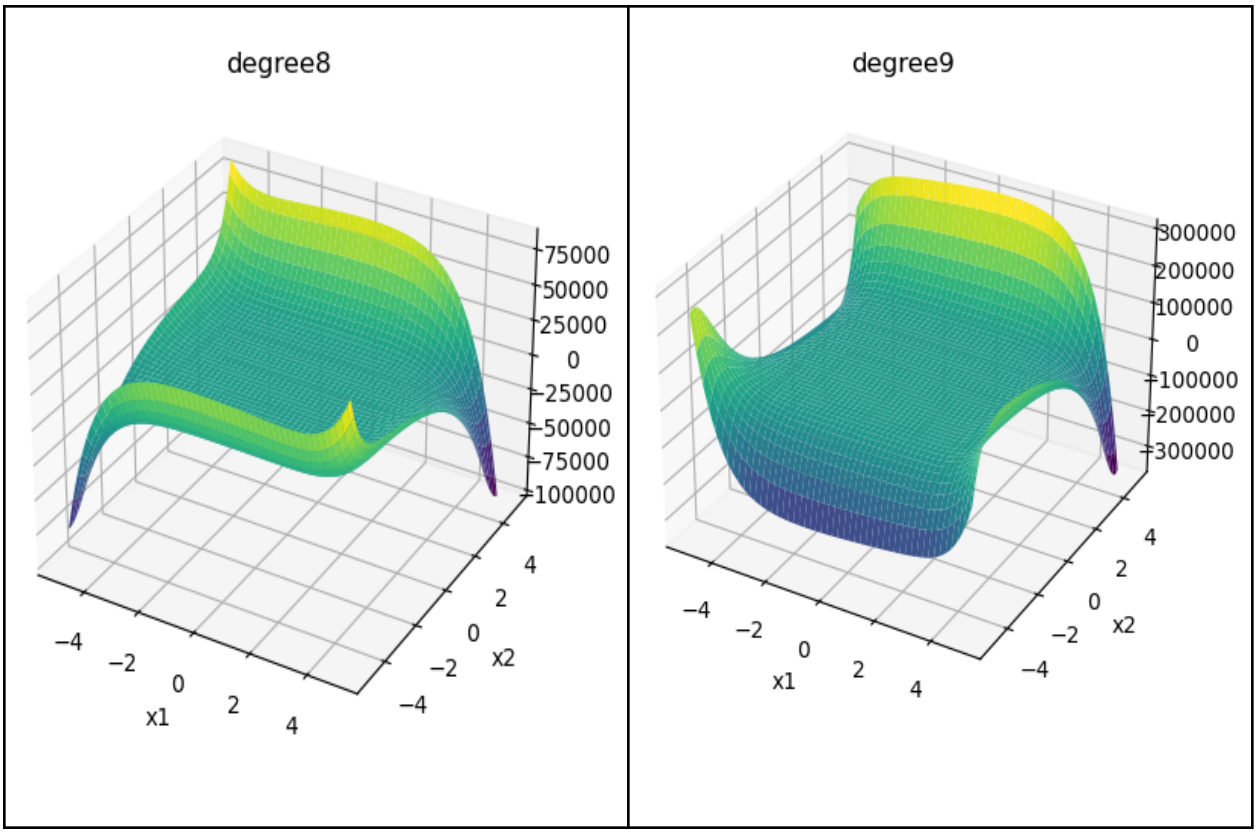
Results Using Stochastic gradient descent, learning rate =0.001

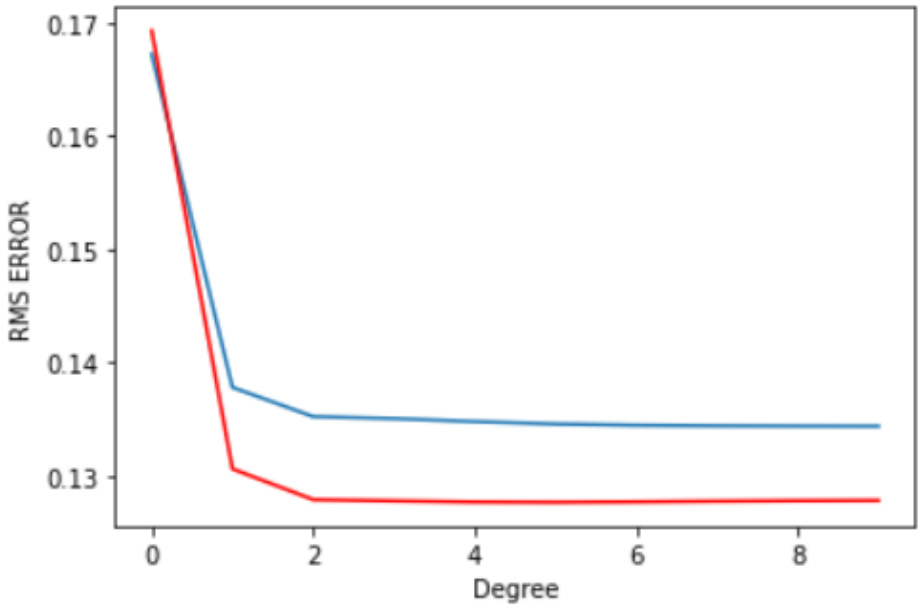
Degree	Training Loss	Testing Loss
0	6.1170113135483753773	1.5545918034469308384
1	4.836717290332821309	1.2214145672013613656
2	4.459647080483491564	1.0910590759808977694
3	4.3725165667437451873	1.0502280235192480896
4	4.324728766983356002	1.0240392675671088472
5	4.321686519034075293	1.0258308840087332669
6	4.3341002387467885155	1.0311761099142340704
7	4.327987060289374112	1.0263959240390452128
8	1.019355895048470077	1.019355895048470077
9	4.3135458714480384144	1.014224918188161381

NOTE: Sometimes Stochasitc gradient descent gives very high values of error, as the process is stochastic, the sampling is based on the randomness.


degree0
0.48
0.47
0.46
0.45
0.44
−4
−2
0
2
4
x1
x2
degree1
4
2
0
−2
−4
degree2
10
5
0
−5
−10
−15
−20
degree3
60
40
20
0
−20
−40
−60


degree4
200
100
0
−100
−200
−300
−4
−2
0
2
4
x1
x2
degree5
1000
500
0
−500
−1000
−1500
degree6
4000
2000
0
−2000
−4000
−6000
degree7
20000
10000
0
−10000
−20000





RMS ERROR vs Degree Plot

Blue line represents the root mean square training error.
Red line represents the root mean square testing error.

Regularization

Now both these descent methods don't put a bound on the growth of coefficients, due to which our training error might decrease, but our testing error will increase as our model cannot capture the generality of the dataset. To curb this overfitting, we do regularization.

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right]$$

$$\min_\theta J(\theta)$$

The last term here is the *regularization factor*, and its value decides how much weightage is to be given to curb the growth of coefficients. The above equation is an example of ridge regression, and we can have different types depending on our selection of the powers of coefficients in the above equation.

Implementation of regularization:

We have implemented regularization using a gradient descent approach with different loss functions with different values of q and different gradients. Regularization factor *lambda* is used.

```
[19] def regularized_loss(y, yhat, w, q, l):
         w_wo = w[1:]
         if q == 2 or q == 4:
           return ( np.sum(np.square(y-yhat)) + l*np.sum(np.power(w_wo,q)) )/2
         if q == 1:
           return ( np.sum(np.square(y-yhat)) + l*np.sum(np.absolute(w_wo)) )/2
         if q == 0.5:
           return ( np.sum(np.square(y-yhat)) + l*np.sum( np.power(np.absolute(w_wo),0.5) ) )/2
```

```
[20] def regularized_gradient(X, w, y, q , l):
       w_wo = w.copy()
       w_wo = w_wo[1:]
       if q == 2:
         return ( (X.T @ ( X@w - y )) + np.insert(l*w_wo, 0, [0], axis=0) )
       if q == 4:
         return ( (X.T @ ( X@w - y )) + np.insert( 2*l*np.power(w_wo,3), 0, [0], axis=0) )
       if q == 1:
         return ( ( X.T @ ( X@w - y ) ) + np.insert( l*np.sign(w_wo)/2, 0, [0], axis=0) )
       if q == 0.5:
         return ( ( X.T @ ( X@w - y ) ) + np.insert( l*np.sign(w_wo)*np.power(np.absolute(w_wo),-0.5)/4 , 0, [0], axis=0) )
```

For q = 0.5

lamba	Min. training error	Min. testing error
0	4.1505076312735202605	0.93097355970930684437
1e-20	4.1505076312735202605	0.93097355970930684437
1e-15	4.1505076312735209158	0.9309735597093075221
1e-10	4.1505076313390839994	0.93097355977709904797
0.0001	4.1505731948602373263	0.9310413516451371425
0.1	4.2159179212451476605	0.9984956507315867342
1	4.7898641857067834797	1.5794482108425117193
2	5.3900287856925013412	2.1484721165211080297
3	5.795826030522143274	2.3488456049626608682

q=1 lasso

lamba	Min. training error	Min. testing error
0	4.150632585468080815	0.93166806214472916046
1e-20	4.150632585468080815	0.93166806214472916046
1e-15	4.1506325854680812676	0.93166806214472964136
1e-10	4.1506325855133669855	0.9316680621928169102
0.0001	4.150677871377496825	0.93171614942429040884
0.1	4.1956573146997178173	0.9792856342897258897
1	4.577350131444493393	1.3655278001486000723
2	4.951779352221000594	1.7053520193833679741
3	5.2739202477976024155	1.9511407198490328659

q=2 Ridge

lamba	Min. training error	Min. testing error
0	4.150632585468080815	0.93166806214472916046
1e-20	4.150632585468080815	0.93166806214472916046
1e-15	4.150632585468081059	0.9316680621447294313
1e-10	4.150632585492526514	0.9316680621718041287
0.0001	4.150657030795355095	0.9316951364505440407
0.1	4.1747128569847696713	0.95809518193676981 37
1	4.3634215242798446406	1.1483646567059552187
2	4.5289446067246299883	1.2909914365144910611
3	4.6627305336883709576	1.390204689797272597

q=4

lamba	Min. training error	Min. testing error
0	4.150632585468080815	0.93166806214472916046
1e-20	4.150632585468080815	0.93166806214472916046
1e-15	4.150632585468080907	0.9316680621447292689
1e-10	4.1506325854772665755	0.93166806215557856506
0.0001	4.1506417709462741336	0.93167891104252738586
0.1	4.159548344951731885	0.94203721589603089454
1	4.222856916698151787	1.0073359756090267005
2	4.273150343653496434	1.0509905758354039283
3	4.3125215175794914036	1.0816260079281685448

Conclusion :

1. Out of all the regression models, we have varied the number of iterations and changed the learning rate to compare the models for each degree. The model with a low value of testing error will be preferred as the optimal model.

2. Upon careful observation, we have found out that the **polynomial of degree 5** has the most negligible testing error and training error values in comparison to other models

3. We can also observe that as the degree increases, the testing error increases (training error is increasing here since we have fixed the number of iterations. If we increase the number of iterations, we can achieve a minimum training error), indicating the overfitting nature of the model to the training data. This can be seen by the visualization through the plots provided too.