A Project Report

On

Building Polynomial Regression with Regularization

BY

Kshitij Verma (2017B1A71145H)

Rahul Prakash (2018AAPS0893H)

Siddharth Raj (2018AAPS0398H)

Under the supervision of

Dr. NL Bhanu Murthy

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(OCTOBER , 2020)

Assignment 3: Implementing Polynomial Regression with Regularization

In this assignment, we have been given an "insaurance.txt" file. We have implemented polynomial regression of varying degree from 1 to 10 using Gradient Descent and Stochastic Gradient Descent Algorithm with L1 (Lasso) and L2 (Ridge) regularization.

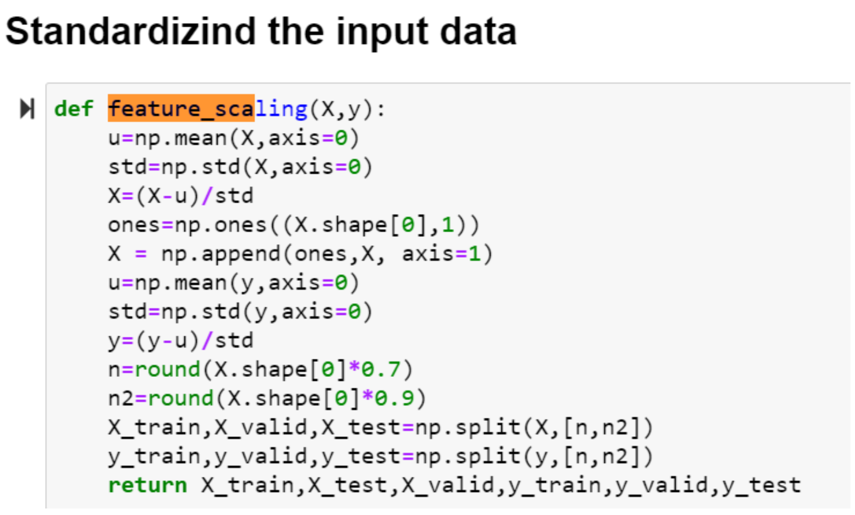
Steps in building the model:

Mentioned below are the steps involved in building the Model. They are:

1. **Data Pre-Processing:** Data Preprocessing is a technique that transforms the given raw data into a clean dataset. This step improves the accuracy of the model. Mostly 2 methods are used for data pre-processing. They are a) Standardization and b) Normalization.
   Normalization: This step is to make all the elements lie between 0 and 1 thus bringing all the values of numeric columns in the dataset to a common scale.
   Standardization: Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1.

   The formula to standardize the data is: $Z = \frac{(xi - \mu)}{\sigma}$

   Standardization is implemented in our code as :



Standardizind the input data

```
def feature_scaling(X,y):
    u=np.mean(X,axis=0)
    std=np.std(X,axis=0)
    X=(X-u)/std
    ones=np.ones((X.shape[0],1))
    X = np.append(ones,X, axis=1)
    u=np.mean(y,axis=0)
    std=np.std(y,axis=0)
    y=(y-u)/std
    n=round(X.shape[0]*0.7)
    n2=round(X.shape[0]*0.9)
    X_train,X_valid,X_test=np.split(X,[n,n2])
    y_train,y_valid,y_test=np.split(y,[n,n2])
    return X_train,X_test,X_valid,y_train,y_valid,y_test
```

1. **Building the model:** The data was read from "Insurance.txt" file. Random shuffling of the data is done each time a regression model is made using the random shuffle function. After random shuffling, the entire dataset is divided into three parts, training set, validation set and testing set. The training set consists of 70% of the dataset, the validation set has 20% of the dataset and the testing set consists of remaining 10% of the dataset.

2. **Using Gradient Descent Method:** The hypothesis function in the model is:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

Here, theta are the weights, x are the independent variable and h(x) is the predicted value. The hypothesis is calculated in the form of matrix multiplication using:

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

The Cost function is calculated as :

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

The weights after each iteration is calculated as :

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j := 0..n}$$

3. **Regularization: Regularization** is a process of introducing additional information in order to prevent overfitting. The focus for this article is L1 and L2 regularisation.

Loss Function Without Regularization:

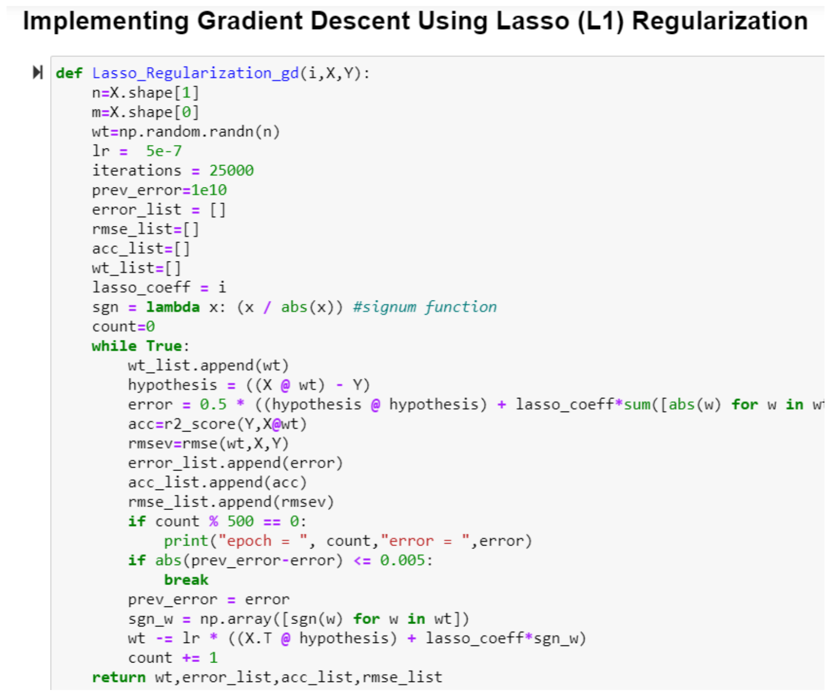
$$Loss = Error(y, \hat{y})$$

Loss Function With L1 (Lasso) Regularization:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss Function With L2 (Ridge) Regularization:

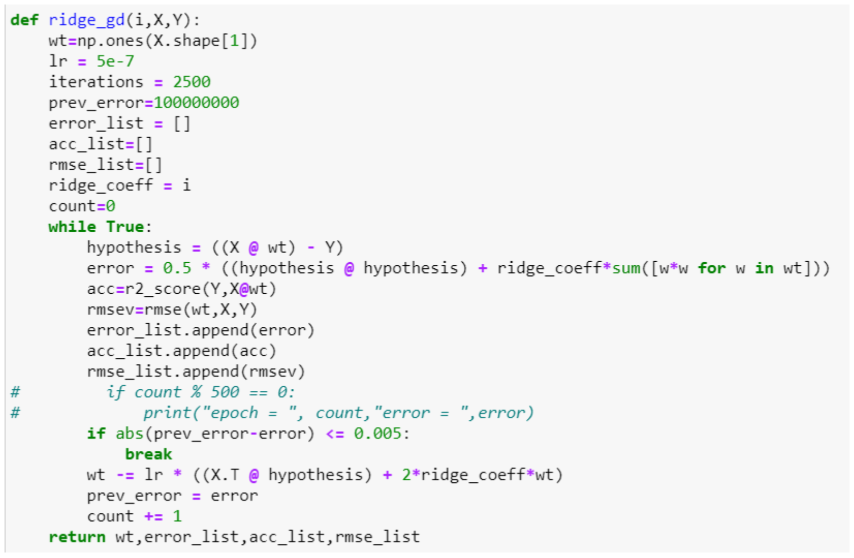
$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$



Implementing Gradient Descent Using Lasso (L1) Regularization

```
def Lasso_Regularization_gd(i,X,Y):
    n=X.shape[1]
    m=X.shape[0]
    wt=np.random.randn(n)
    lr =  5e-7
    iterations = 25000
    prev_error=1e10
    error_list = []
    rmse_list=[]
    acc_list=[]
    wt_list=[]
    lasso_coeff = i
    sgn = lambda x: (x / abs(x)) #signum function
    count=0
    while True:
        wt_list.append(wt)
        hypothesis = ((X @ wt) - Y)
        error = 0.5 * ((hypothesis @ hypothesis) + lasso_coeff*sum([abs(w) for w in w
        acc=r2_score(Y,X@wt)
        rmsev=rmse(wt,X,Y)
        error_list.append(error)
        acc_list.append(acc)
        rmse_list.append(rmsev)
        if count % 500 == 0:
            print("epoch = ", count,"error = ",error)
        if abs(prev_error-error) <= 0.005:
            break
        prev_error = error
        sgn_w = np.array([sgn(w) for w in wt])
        wt -= lr * ((X.T @ hypothesis) + lasso_coeff*sgn_w)
        count += 1
    return wt,error_list,acc_list,rmse_list
```

Fig. Gradient Descent with L1 Regularization



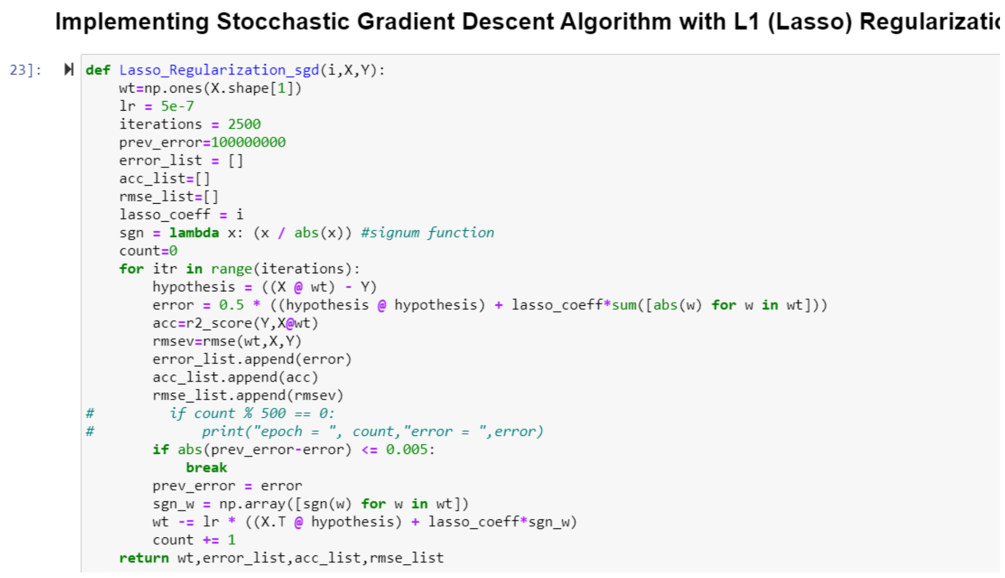
```
def ridge_gd(i,X,Y):
    wt=np.ones(X.shape[1])
    lr = 5e-7
    iterations = 2500
    prev_error=100000000
    error_list = []
    acc_list=[]
    rmse_list=[]
    ridge_coeff = i
    count=0
    while True:
        hypothesis = ((X @ wt) - Y)
        error = 0.5 * ((hypothesis @ hypothesis) + ridge_coeff*sum([w*w for w in wt]))
        acc=r2_score(Y,X@wt)
        rmsev=rmse(wt,X,Y)
        error_list.append(error)
        acc_list.append(acc)
        rmse_list.append(rmsev)
#         if count % 500 == 0:
#             print("epoch = ", count,"error = ",error)
        if abs(prev_error-error) <= 0.005:
            break
        wt -= lr * ((X.T @ hypothesis) + 2*ridge_coeff*wt)
        prev_error = error
        count += 1
    return wt,error_list,acc_list,rmse_list
```

Fig. Gradient Descent with L2 (Ridge Regularization)

4. **Stochastic Gradient Method:** SGD it picks up a "random" instance of training data at each step and then computes the gradient making it much faster as there is much fewer data to manipulate at a single time, unlike Batch GD.

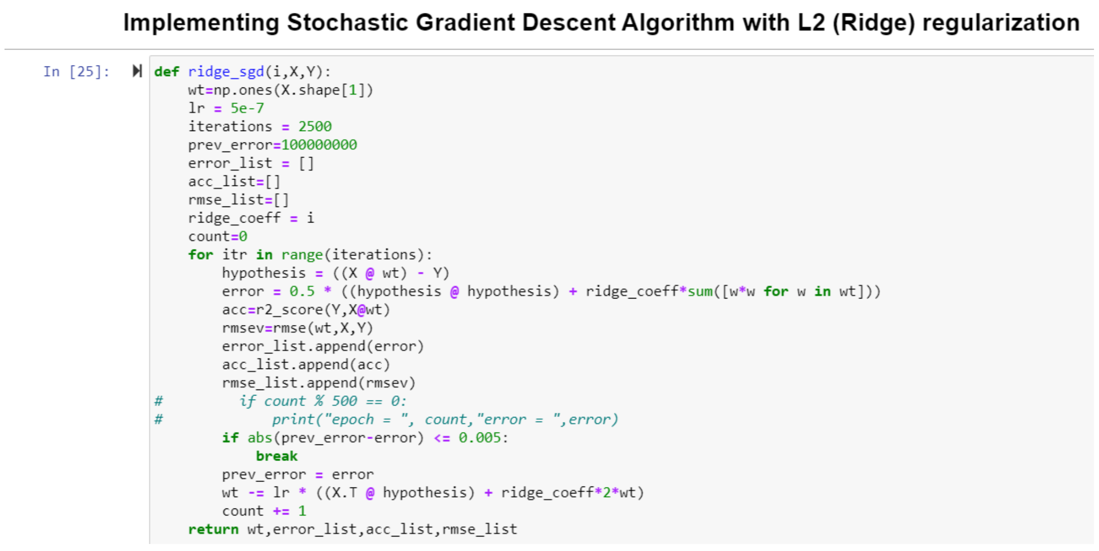
$$\theta_j = \theta_j - \alpha\,(\widehat{y^i} - y^i)\,X_j^i$$

Fig. Stochastic Gradient Descent Algorithm


Implementing Stocchastic Gradient Descent Algorithm with L1 (Lasso) Regularizati

```
23]:   def Lasso_Regularization_sgd(i,X,Y):
           wt=np.ones(X.shape[1])
           lr = 5e-7
           iterations = 2500
           prev_error=100000000
           error_list = []
           acc_list=[]
           rmse_list=[]
           lasso_coeff = i
           sgn = lambda x: (x / abs(x)) #signum function
           count=0
           for itr in range(iterations):
               hypothesis = ((X @ wt) - Y)
               error = 0.5 * ((hypothesis @ hypothesis) + lasso_coeff*sum([abs(w) for w in wt]))
               acc=r2_score(Y,X@wt)
               rmsev=rmse(wt,X,Y)
               error_list.append(error)
               acc_list.append(acc)
               rmse_list.append(rmsev)
       #         if count % 500 == 0:
       #             print("epoch = ", count,"error = ",error)
               if abs(prev_error-error) <= 0.005:
                   break
               prev_error = error
               sgn_w = np.array([sgn(w) for w in wt])
               wt -= lr * ((X.T @ hypothesis) + lasso_coeff*sgn_w)
               count += 1
           return wt,error_list,acc_list,rmse_list
```

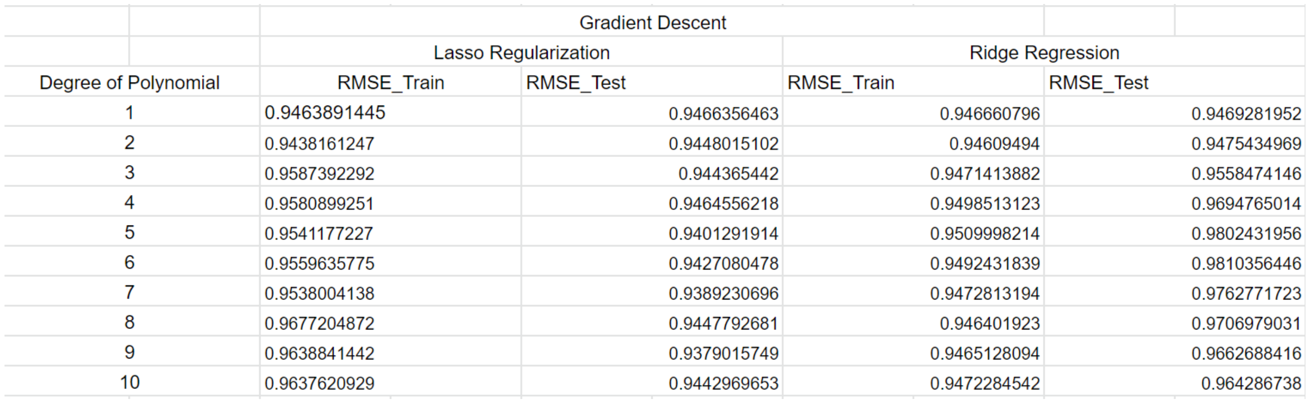
Fig. Stochastic Gradient Descent Algorithm with L1 (Lasso) Regularization


Implementing Stochastic Gradient Descent Algorithm with L2 (Ridge) regularization

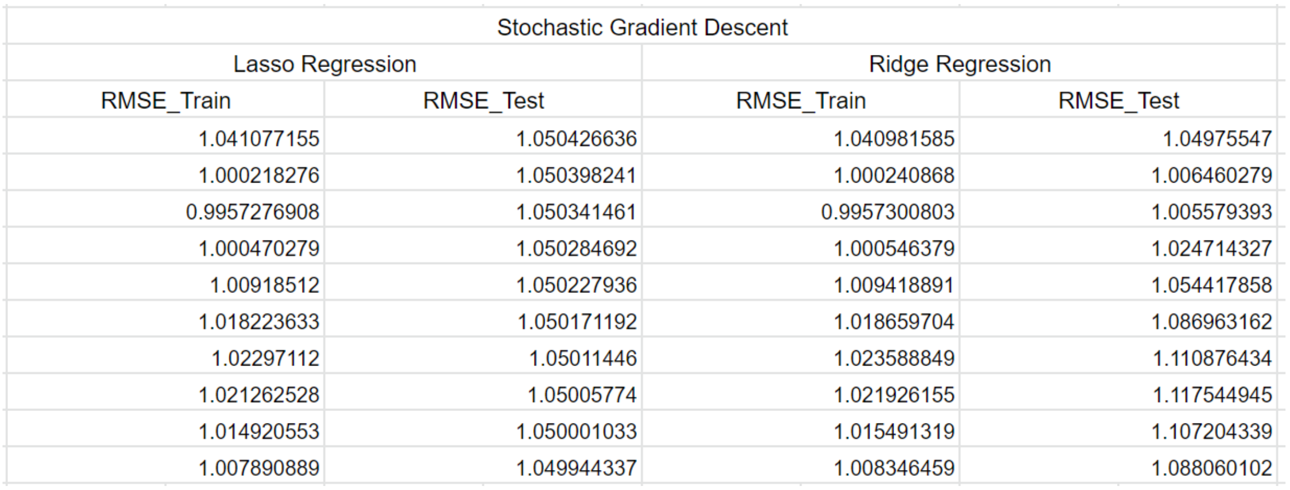
```
In [25]:   def ridge_sgd(i,X,Y):
               wt=np.ones(X.shape[1])
               lr = 5e-7
               iterations = 2500
               prev_error=100000000
               error_list = []
               acc_list=[]
               rmse_list=[]
               ridge_coeff = i
               count=0
               for itr in range(iterations):
                   hypothesis = ((X @ wt) - Y)
                   error = 0.5 * ((hypothesis @ hypothesis) + ridge_coeff*sum([w*w for w in wt]))
                   acc=r2_score(Y,X@wt)
                   rmsev=rmse(wt,X,Y)
                   error_list.append(error)
                   acc_list.append(acc)
                   rmse_list.append(rmsev)
           #         if count % 500 == 0:
           #             print("epoch = ", count,"error = ",error)
                   if abs(prev_error-error) <= 0.005:
                       break
                   prev_error = error
                   wt -= lr * ((X.T @ hypothesis) + ridge_coeff*2*wt)
                   count += 1
               return wt,error_list,acc_list,rmse_list
```

Fig. Stochastic Gradient Descent Algorithm with L2 (Ridge) regularization

Error Obtained for Training and Testing for different Methods



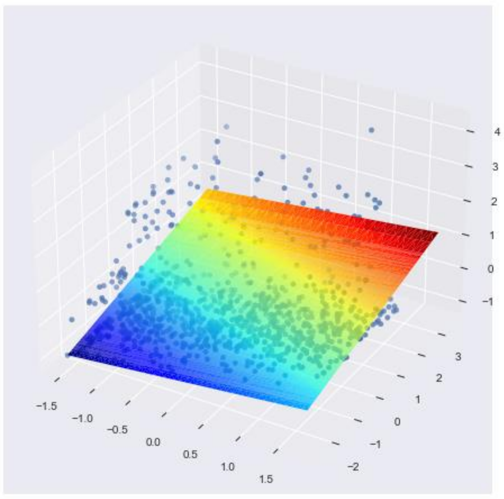
	Gradient Descent			
	Lasso Regularization		Ridge Regression	
Degree of Polynomial	RMSE_Train	RMSE_Test	RMSE_Train	RMSE_Test
1	0.9463891445	0.9466356463	0.946660796	0.9469281952
2	0.9438161247	0.9448015102	0.94609494	0.9475434969
3	0.9587392292	0.944365442	0.9471413882	0.9558474146
4	0.9580899251	0.9464556218	0.9498513123	0.9694765014
5	0.9541177227	0.9401291914	0.9509998214	0.9802431956
6	0.9559635775	0.9427080478	0.9492431839	0.9810356446
7	0.9538004138	0.9389230696	0.9472813194	0.9762771723
8	0.9677204872	0.9447792681	0.946401923	0.9706979031
9	0.9638841442	0.9379015749	0.9465128094	0.9662688416
10	0.9637620929	0.9442969653	0.9472284542	0.964286738

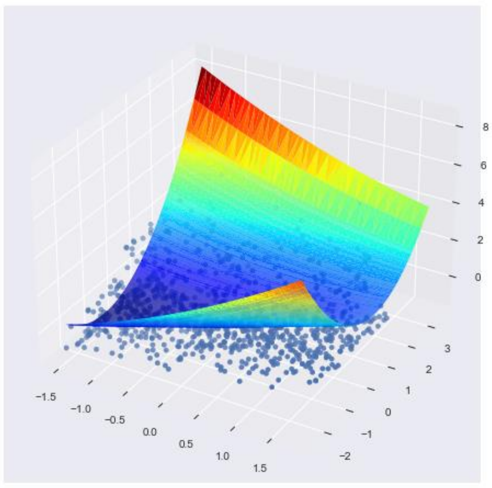


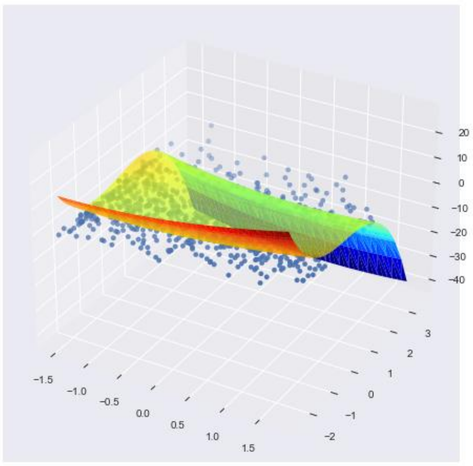
Stochastic Gradient Descent			
Lasso Regression		Ridge Regression	
RMSE_Train	RMSE_Test	RMSE_Train	RMSE_Test
1.041077155	1.050426636	1.040981585	1.04975547
1.000218276	1.050398241	1.000240868	1.006460279
0.9957276908	1.050341461	0.9957300803	1.005579393
1.000470279	1.050284692	1.000546379	1.024714327
1.00918512	1.050227936	1.009418891	1.054417858
1.018223633	1.050171192	1.018659704	1.086963162
1.02297112	1.05011446	1.023588849	1.110876434
1.021262528	1.05005774	1.021926155	1.117544945
1.014920553	1.050001033	1.015491319	1.107204339
1.007890889	1.049944337	1.008346459	1.088060102

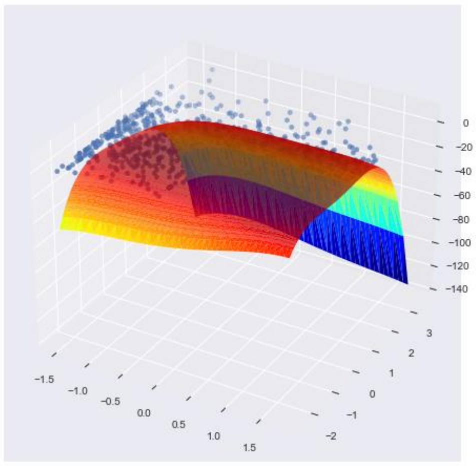
	Gradient Descent		Stochastic Gradient Descent	
	Lasso Regularization	Ridge Regularization	Lasso Regularization	Ridge Regularization
Degree of Polynomial	RMSE_Validate	RMSE_Validate	RMSE_Validate	RMSE_Validate
1	0.9197166722	0.9226234348	1.011888013	1.011323492
2	0.9226931301	0.9241208636	1.011862992	0.9752621564
3	0.9252306533	0.9250826483	1.011812957	0.9726179316
4	0.9220056323	0.9263628952	1.011762935	0.9767103112
5	0.9294530192	0.9248169775	1.011712925	0.9840098984
6	0.9104278214	0.9217814353	1.011662926	0.9909864786
7	0.9173141002	0.9219298954	1.011612939	0.9928637495
8	0.9408278171	0.9253819692	1.011562963	0.9882263178
9	0.9316363336	0.9307650979	1.011513	0.981232367
10	0.9335309821	0.9362152215	1.011463048	0.9776167679

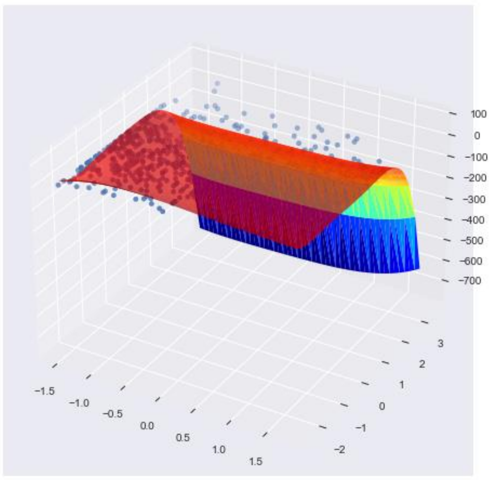
Surface Graphs for Different Degree Polynomial

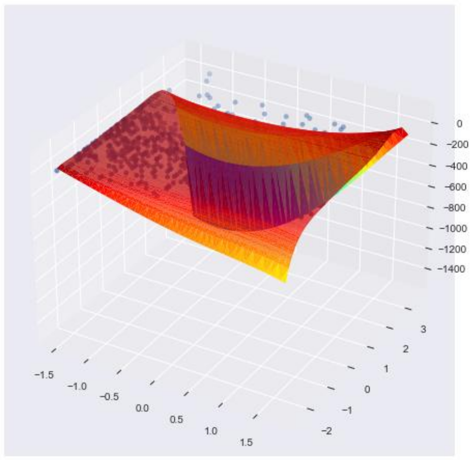

Degree 1

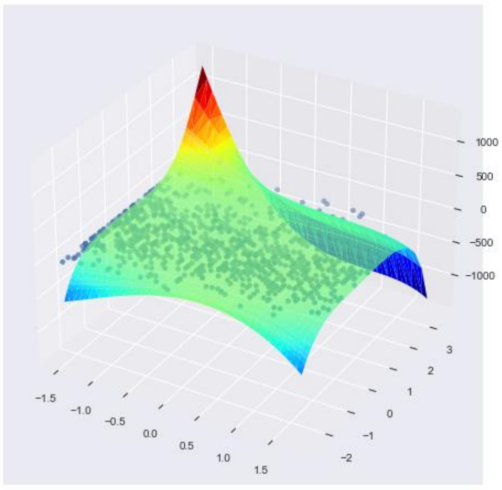

Degree 2


Degree 3

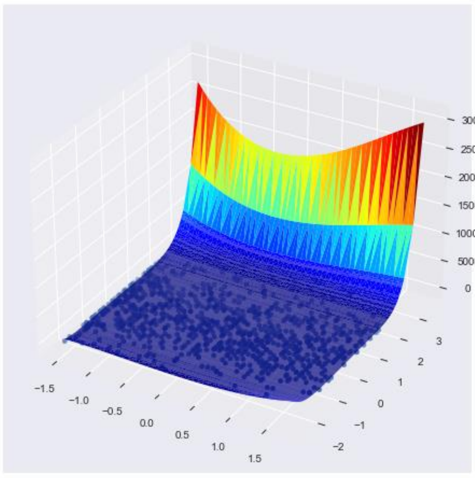

Degree 4


Degree 5

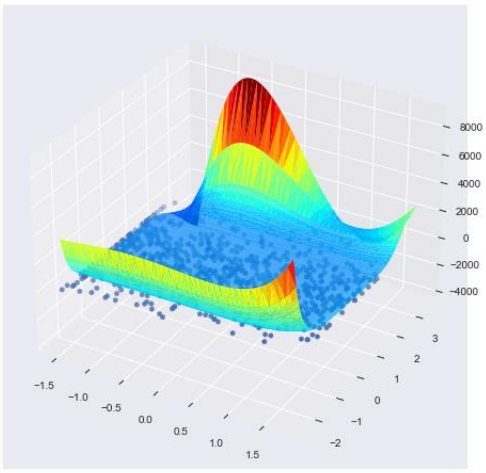

Degree 6



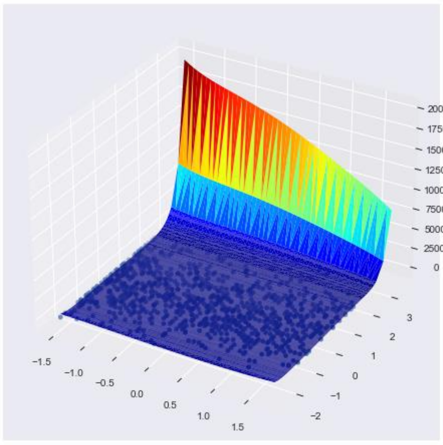
Degree 7



Degree 8



Degree 9



Degree 10

A polynomial of a higher degree will cause more overfitting. As a result, the error shown by the polynomial of higher degree for testing data is very high.

Questions to Ponder On

1. **What happens to the training and testing error as polynomials of higher degree are used for prediction?**

Ans) As the flexibility in the model increases (by increasing the polynomial degree) the training error continually decreases due to increased flexibility. However, the error on the testing set only decreases as we add flexibility up to a certain point. As the flexibility increases beyond this point, the testing error increases because the model has memorized the training data and the noise. Both overfitting and underfitting can be realized and removed by understanding the importance of model evaluation and optimization using cross-validation.

2. **Does a single global minimum exist for Polynomial Regression as well? If yes, justify.**

Ans) Yes, global minima exists in case of polynomial regression. In case of polynomial regression, before a certain degree of polynomial, the training data is high while after that degree, the testing data is higher.

Hence the error is minimum only for a certain degree of polynomial. Plotting a curve for Least Square errors vs degree of polynomial also demonstrates this fact as it is turns out to be a parabola.

3. **Which form of regularization curbs overfitting better in your case? Can you think of a case when Lasso regularization works better than Ridge?**

Ans) We observed that Ridge regularization showed slightly better results as compared to lasso regularization.

b. Lasso tends to do well if there are a small number of significant parameters and the others are close to zero (when only a few predictors actually influence the response). Ridge works well if there are many large parameters of about the same value (when most predictors impact the response).

4. **How does the regularization parameter affect the regularization process and weights? What would happen if a higher value for λ (> 2) was chosen?**

Ans) As the value of regularization parameter increases it reduces the value of coefficients and thus reduces the variance without losing the important properties in the data. We need to find an optimal value of λ so that the generalization error is small.

λ is a hyper parameter whose value is decided by us. If **λ(>2)** is high it adds high penalty to error term making the learned hyper plain almost linear, if **λ** is close to 0 it has almost no effect on the error term causing no regularization. Hence, for higher value of **λ ,** algorithm even fails to fit training data and result in **underfitting**. Also gradient descent will fail to converge.

5. **Regularization is necessary when you have a large number of features but limited training instances. Do you agree with this statement?**

Ans) Regularization, significantly reduces the variance of the model, without substantial increase in its bias.

Adding many new features gives us more expressive models which are able to better fit our training set. If too many new features are added, this can lead to overfitting of the training set. Hence, regularization is necessary to zero-out some features in order to avoid overfitting and/or to reduce the computational complexity.

6. **If you are provided with D original features and are asked to generate new matured features of degree N, how many such new matured features will you be able to generate? Answer in terms of N and D.**

Ans) Number of features for (D,N)= (D+N)C(N) where D is the number of the original features, N is the degree of the polynomial.

7. **What is bias-variance trade off and how does it relate to overfitting and regularization?**

Ans) The bias – variance tradeoff is explained as follows :

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. This tradeoff in complexity is why there is a tradeoff between bias and variance. To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error.

Overfitting happens when our model captures the noise along with the underlying pattern in data. It happens when we train our model a lot over noisy dataset. These models have low bias and high variance.

The regularized model has a bit higher training error (higher bias) than the polynomial fit but the testing error is greatly improved. The goal that is to avoid fitting the random noise and thus eliminating the high variance issue can be achieved by the bias-variance tradeoff. Thus, the bias-variance tradeoff can be leveraged to improve model predictive capability.