CS60050 - MACHINE LEARNING ASSIGNMENT 1
LINEAR REGRESSION and GRADIENT DESCENT

NAME: **Himanshu Mundhra**
ROLL No. : **16CS10057**

X - The Input Vector(s)
Y - The Corresponding Actual Value
Θ - Parameters or Coefficients of the Input Vector **X**
h(X:Θ) - Estimated Value using Θ and **X**
MeanSquare - The Common MeanSquare **J(Θ)** Cost Function
MeanAbsolute - The **J(Θ)** Cost Function with Absolute Difference instead of Squares
MeanBiquadratic - The **J(Θ)** Cost Function with Biquadratic Difference

In this assignment, we aim to train a machine learning model using linear regression and stochastic gradient descent in order to form a curve **y = sin(2πx) + n(0, 0.3).**
We have used the NumPy, PANDAS, Matplotlib.Pyplot Libraries for this Assignment.
Each part has been made in a separate folder and has been coded in such a way that it can be executed independently of the other parts. Each Part is a Superset of the previous and the Subset of the Next Part.
The **Parameters Learned** and the **Training, Testing Errors** and their variations with the Number of Features, are printed in the form of a Panda.DataFrame in '**Result.txt'** in each local directory, along with the various plots for each different part.with appropriate names.

PART 1: Synthetic data generation and simple curve fitting :

Generation of synthetic dataset :

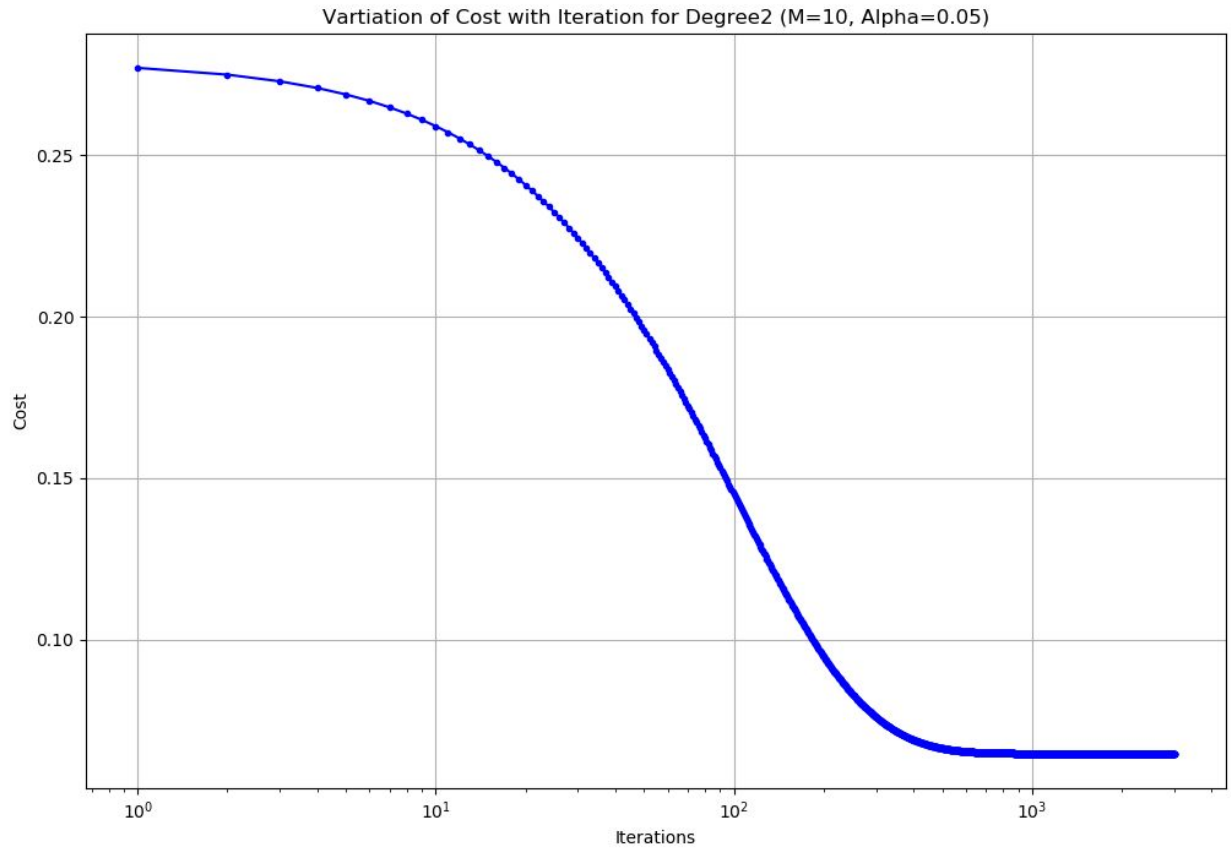
The function **GenerateDataset(M)** performs this task of creating a dataset of size M.
Generation of .**X ~U[0,1)** is done by the *np.random.rand()* function in the NumPy Library.
Generation of **Y~ sin(2πX) + n(0, 0.3)** is done by the *np.sin()* and *np.random.randn()* function in the NumPy Library.

Splitting of synthetic dataset :

The function **Split(Dataset, Training_Fraction)** performs this task of splitting a dataset into a Training Set with the size determined by Training_Fraction.
The *np.shuffle()* of the NumPy Library is used for shuffling, after which we take a slice and split the shuffled dataset into Training and Testing DataSets which are sorted by their X-Coordinates With the help of *argsort()*.

Fitting curve using gradient descent on Mean Square Error function :

The functions **h() , error() , cost() , descent()** help us in carrying out the gradient descent.
The number of iterations has a set value of 3000 which was chosen after some trial and errors and analysing the **Cost v/s Iterations** log-scaled graph, which gave that the value of the Cost Function Saturated at around 3000 Iterations.
We use the same synthesized dataset for all the different values of N (degree of polynomial to be fitted) and initialize the Θ-Vector as **n(0,0.00001).** We choose this value to be such so as to not encounter overflow errors later in the assignment(when we deal with smaller and larger Learning Rates and Different Cost Functions). We update Θ, ITERATION number of times and learn the optimal value of the Θ-Parameter Vector.
This updation is performed on the whole Θ-Parameter Vector at once and for the whole dataset together -> **Stochastic Gradient Descent**
The different values of Training and Testing Error have been recorded along with the Parameters for each Degree Polynomial and stored printed PANDA dataframes in Result.txt



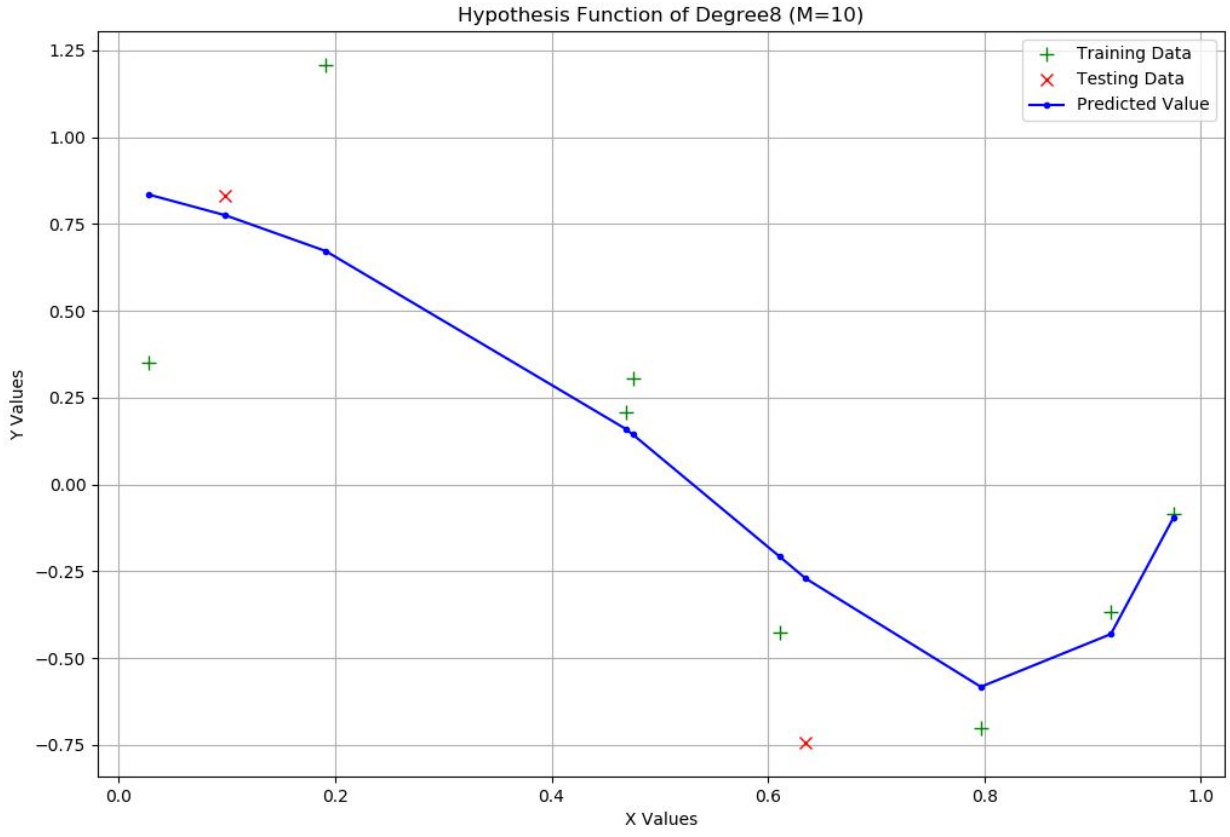
A Sample Plot of the Cost Function vs Iteration(here for N=2) which shows saturation at around the 1000 iterations mark. The other plots safely saturate at around 3000 iterations.

PART 2 : Visualisation of the dataset and the fitted curves :

In this part, we have generated the plots corresponding to the curves we obtained in Part 1 using **matplotlib.pyplot**. This part has been implemented exclusively of Part1 and has all the functionalities of Part1.

Plots of Curves obtained :

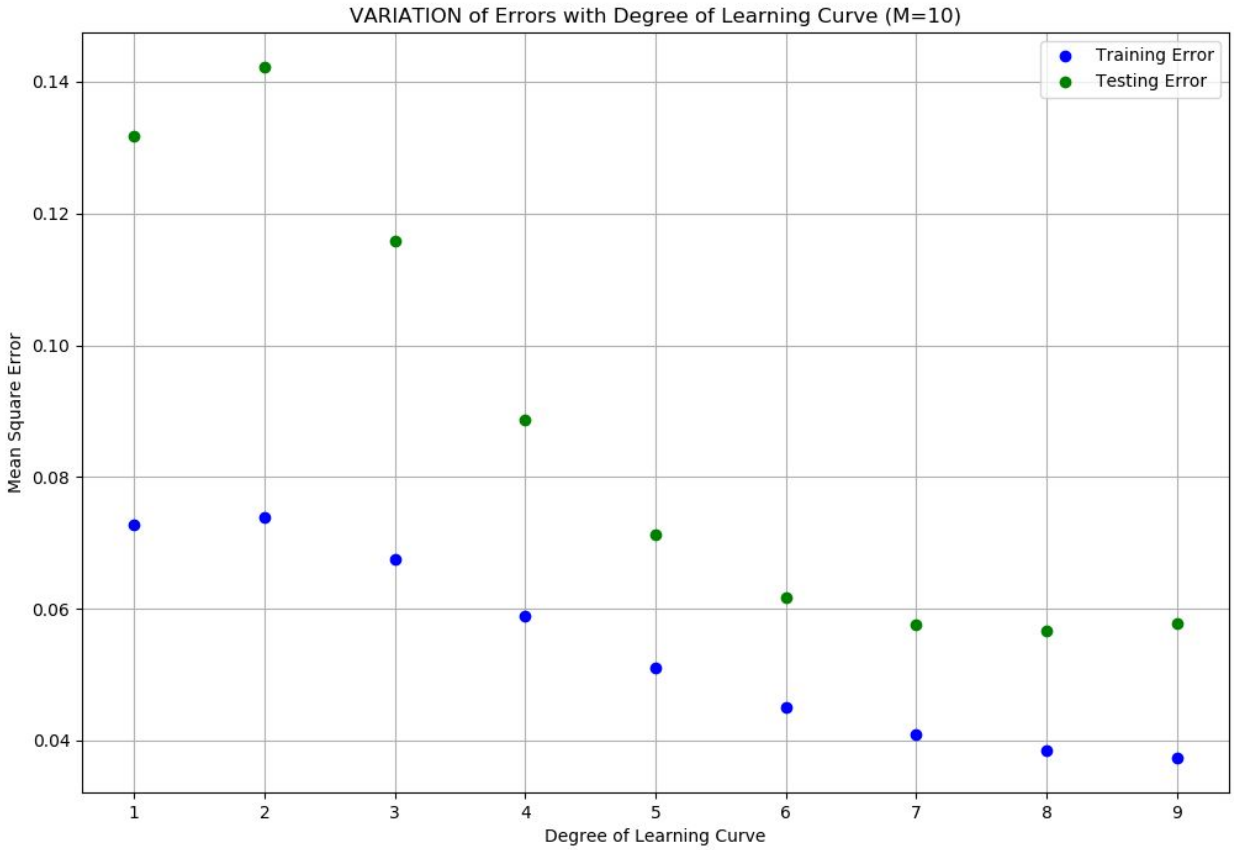
We proceed similarly to Part1 and simply plot the curves with the learned parameters for each value of N. As is expected, the curves with higher degrees have a better visual fit to the Dataset as compared to the curves with lower degrees.
Because of the Dataset being small (10 points) for this part, the plots obtained are not really smooth and are more of a broken line segment.



A Sample Plot of the Hypothesis Function with Degree8 for (M=10). The graph obtained is more of a broken fragmented curve because of the small number of data points we have.

Plot of Error vs Degree of Polynomial :

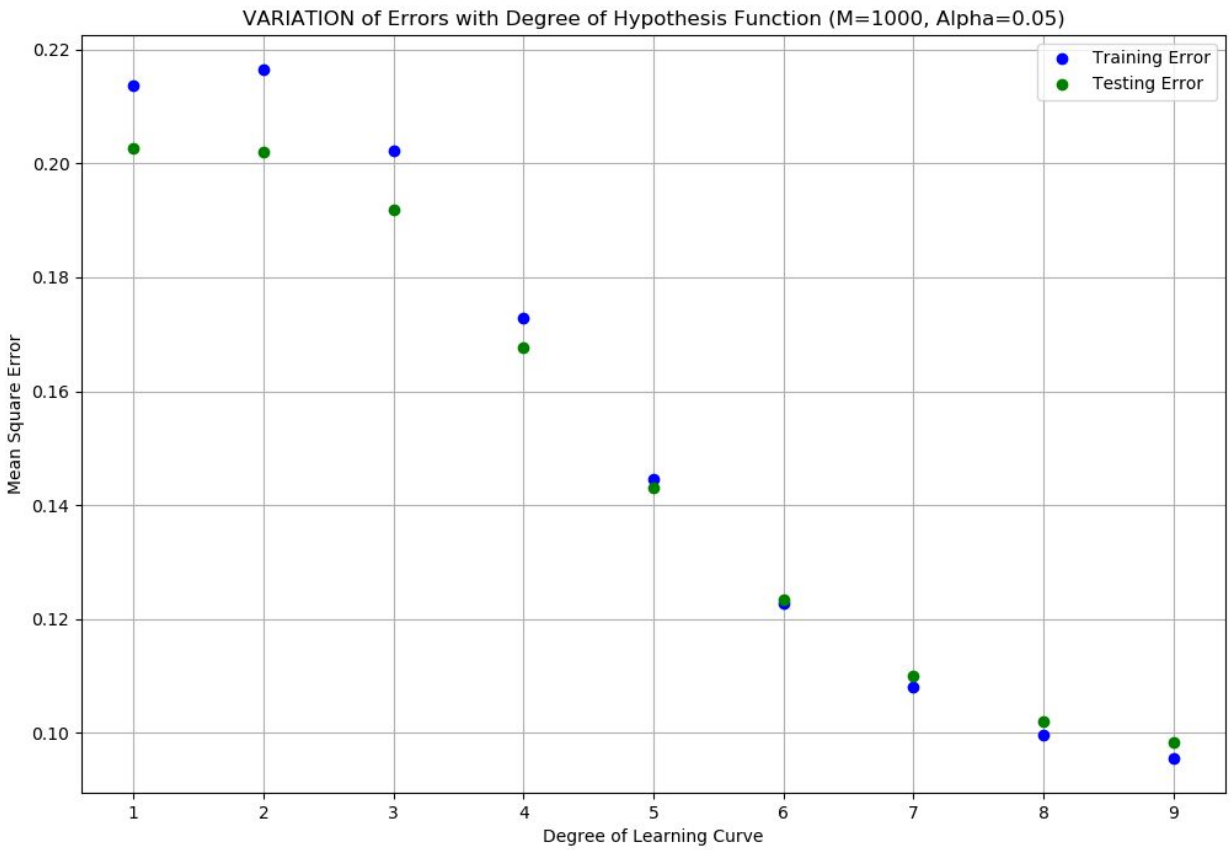
We plot this graph so as to analyse the overfitting and underfitting aspects of curve fitting.

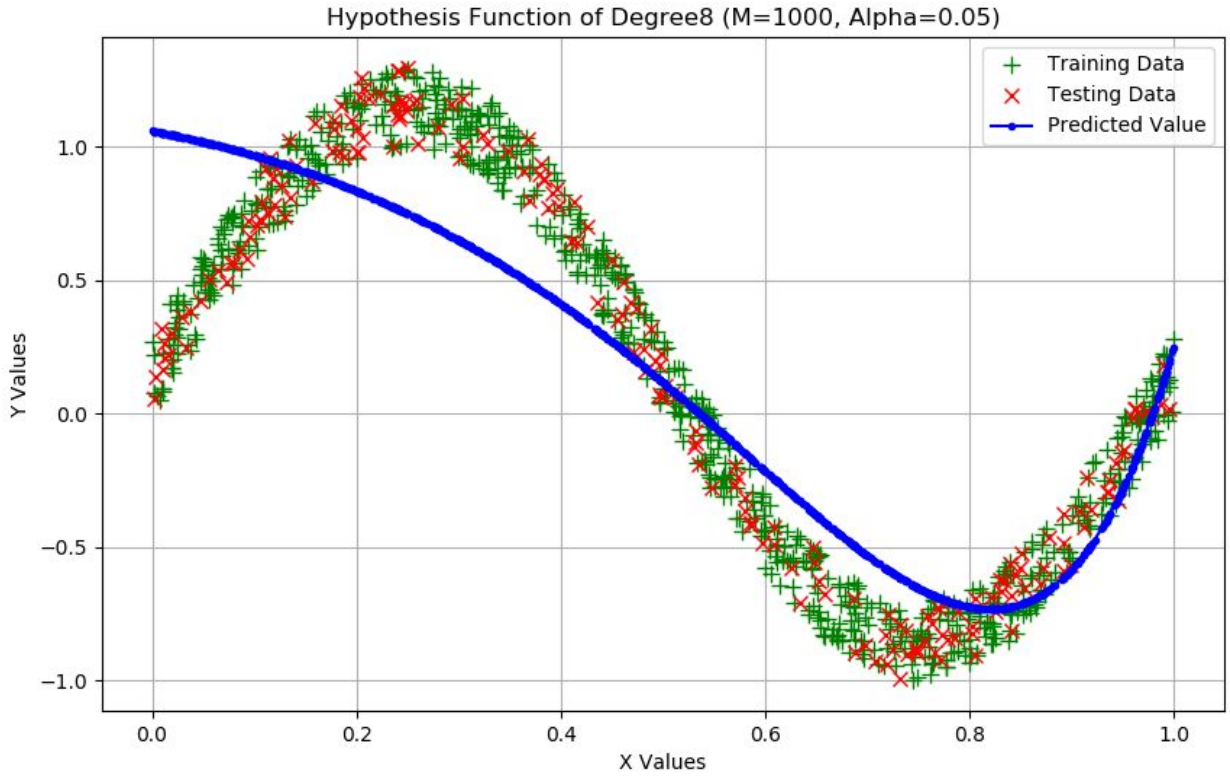


As we observe, the Training and Testing Error are decently high for smaller values of N which imply an underfit of the curve. The values are keeping on decreasing as we can see.
The optimal choice of a hyperparameter such as Degree of Polynomial is usually done with the help of a validation set, but here because we were instructed to use Test Error to determine the value, I have chosen my Test Set to be the Validation Set for the assignment.
Hence the lowest error on the Validation Set is at N=8, which has decently small Training Error as well. Also, we can almost feel the graph to be rising towards higher testing error and falling to lower training error if we increase the value of N.
Hence we obtain the **Optimal Value of our HyperParameter, N = 8.**

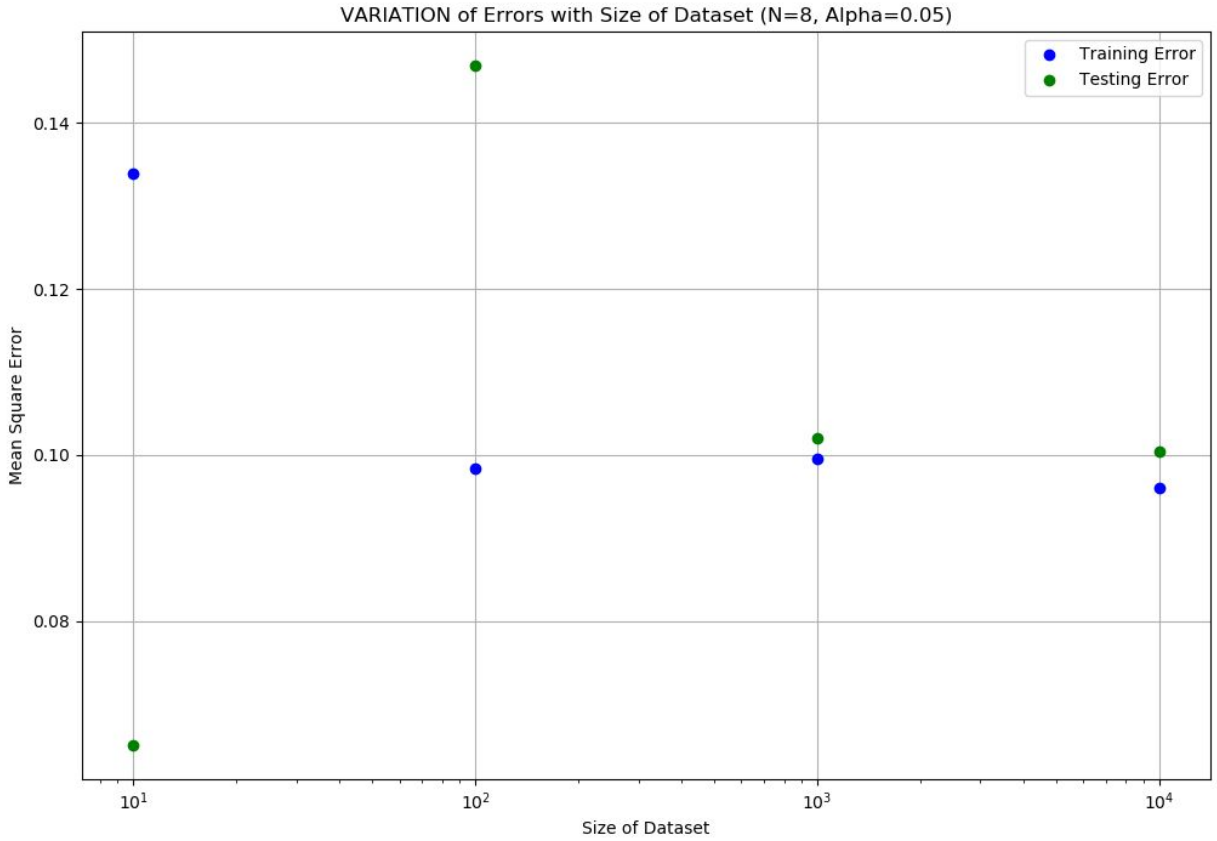
PART 3 : Experimenting with larger training set :

We perform the exact same functions as Part2 except that now we do it in a loop which takes into account the different values of M as well.
This part as well is a logical extension of Part1 and Part2 and can be run independently of them as asked in the question.
We plot various different curves for the data we have obtained and try to analyse their features.
The first thing we notice is the curves get smoother and the data starts to get fit better.


VARIATION of Errors with Degree of Hypothesis Function (M=1000, Alpha=0.05)
Training Error
Testing Error
Mean Square Error
Degree of Learning Curve
0.22
0.20
0.18
0.16
0.14
0.12
0.10
1
2
3
4
5
6
7
8
9


Hypothesis Function of Degree8 (M=1000, Alpha=0.05)
Training Data
Testing Data
Predicted Value
Y Values
X Values
1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

Here we observe that the training and testing error almost coincide for our optimal N = 8 value, a similar thing is observed in the plot for (M=10000). Moreover with the increase in dataset size, the curves we obtain also start to fit better to the synthesized dataset, as is visible from the plot of the estimated function. The increase in the number of datasets always leads to better values in the sense that the randomness and noise is more or less balanced out by a greater number of points and hence a uniformity is obtained in the datasets, which is then learned by the parameters of out hypothesis functions.



This is another plot which shows us the enhancing effect an increase in dataset size brings about to the model. This is a perfectly theoretically expected curve, with Training Error increasing and Testing Error decreasing with an increase in dataset size (dataset quality). The smaller values of M render the model under-fitted and the higher values would result in an over-fitting as can probably be anticipated by the slope of the curve at the end of the above plot.
This plot is for our optimal N value of 8 as obtained in the previous sections. And it is best to have the M value to be equal to 1000 as these curves and the other curves show less of an error at his value of M and smoother fitting.

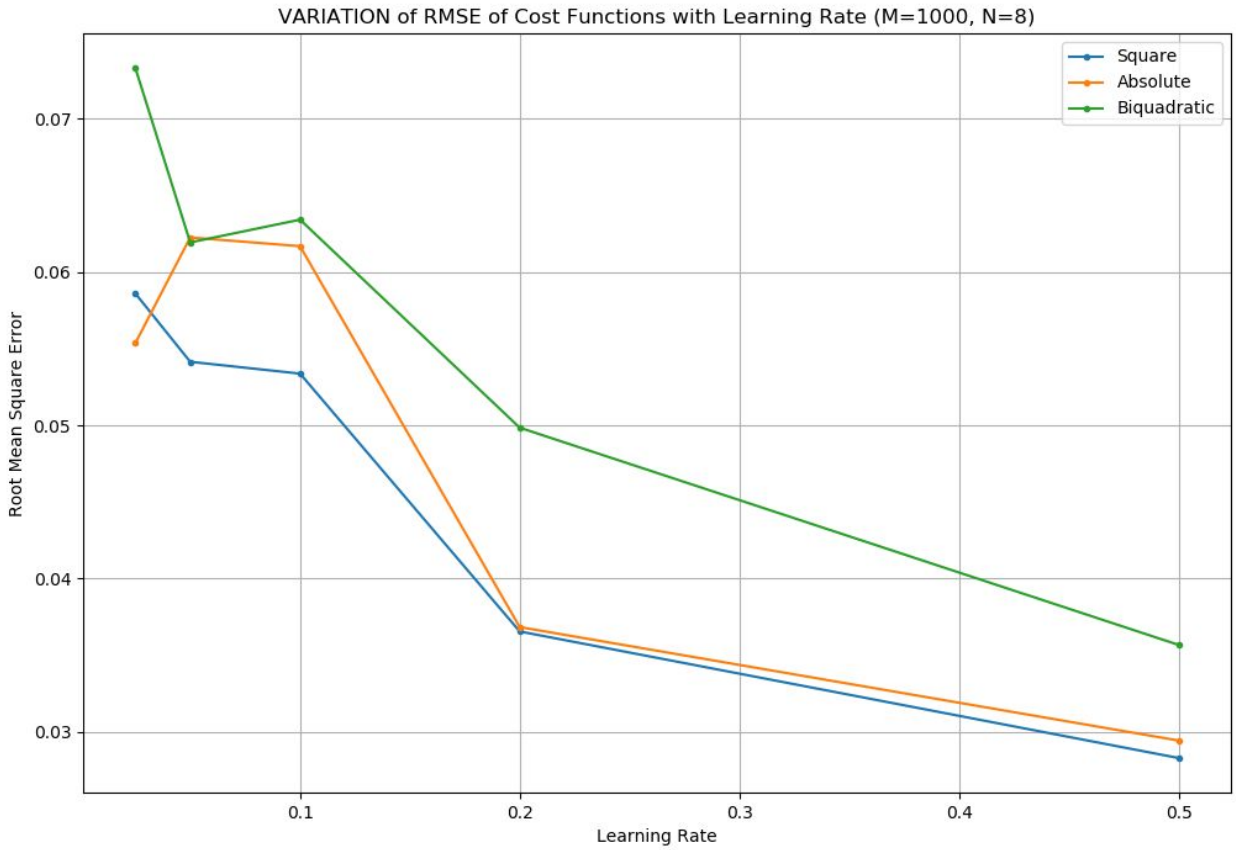
PART 4: Experimenting with cost functions :

Using different Cost Functions :

Here we experiment with different cost functions, the Mean Absolute Difference and the Mean Biquadratic Difference.

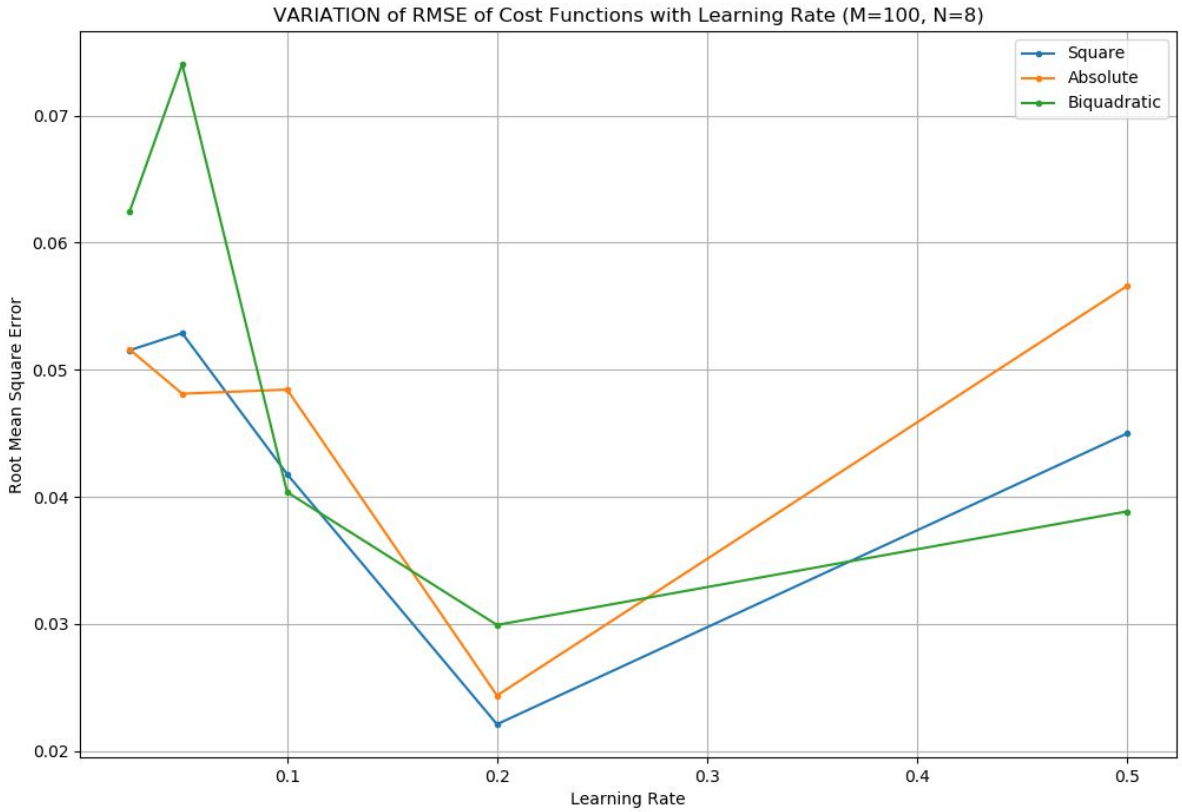
The gradient descent and cost methods were modified to incorporate these two different cost functions in them, switching on the basis of an index sent.

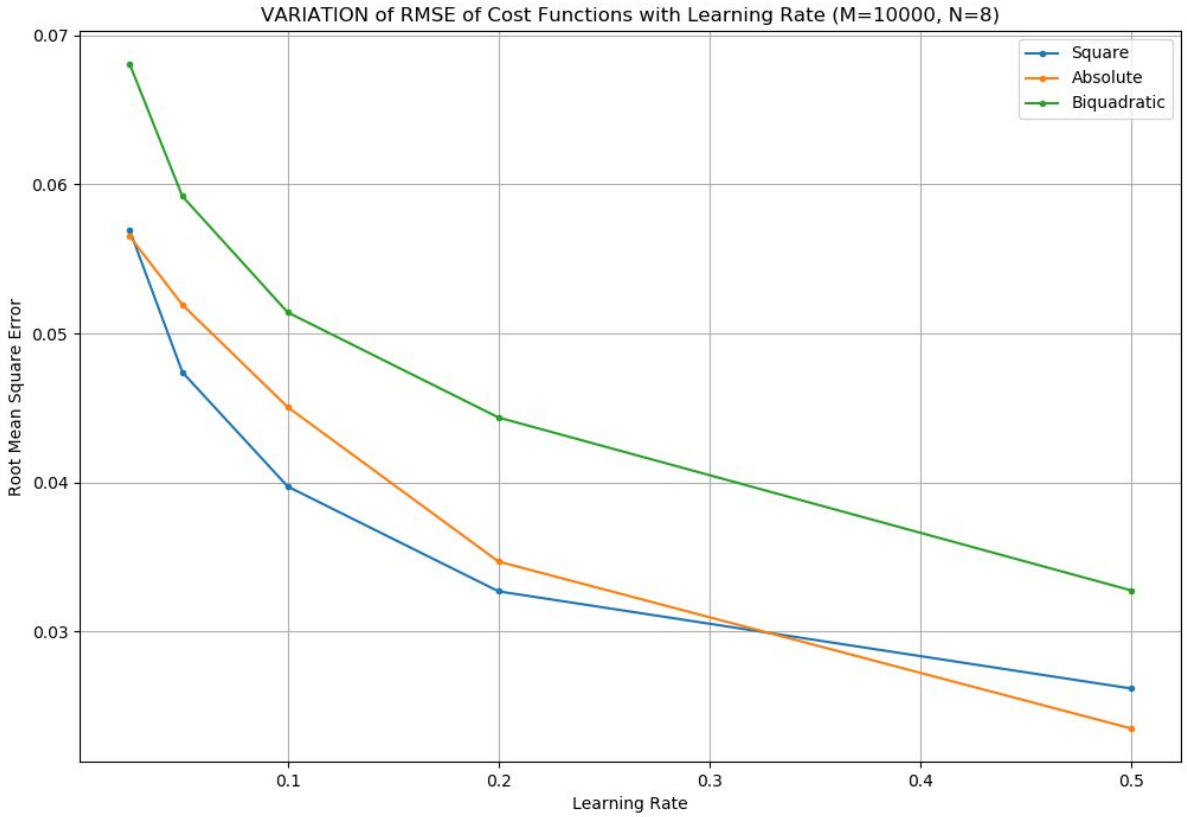
Plotting RMSE vs Learning Rates :



This plot describes the variation of Root Mean Square Error of MeanSquare, MeanAbsolute, MeanBiquadratic Cost Functions with the different Learning Rates [0.025, 0.05, 0.1, 0.2, 0.5].
The trend shouting out is that the Error decreases sharply with an increase in Learning Rate for this particular combination of (M = 1000, N = 8)
For certain other plots, we see a different trend whereby one of the intermediate learning rates is the one with the least error.
Also, in general, most plots have the MeanSquare Cost Function to be the one with the least error in the majority of points which justifies the reason why it is the best-suited cost function for linear regression, gradient descent.
The MeanAbsolute Cost Function loses out due to its non-trivial differentiation which might render it difficult to process when it comes to larger more complex data sets.
Though its performance is quite commendable, almost comparable to the MeanSquare
The MeanBiquadratic has a higher cost factor associated with it as compared to the other two.

But what is the reason that we observe such a disparity?


VARIATION of RMSE of Cost Functions with Learning Rate (M=100, N=8)
Square
Absolute
Biquadratic
Root Mean Square Error
0.07
0.06
0.05
0.04
0.03
0.02
0.1
0.2
0.3
0.4
0.5
Learning Rate


VARIATION of RMSE of Cost Functions with Learning Rate (M=10000, N=8)
Square
Absolute
Biquadratic
Root Mean Square Error
0.07
0.06
0.05
0.04
0.03
0.1
0.2
0.3
0.4
0.5
Learning Rate

The main reason for the MeanSquare Cost Function to be the best Cost Function has a probabilistic and a statistical pre-cursor to it.

For any **Y,** we can say that

$$Y = h(X:\Theta) + \varepsilon$$
$$Y - h(X:\Theta) = \varepsilon$$

In statistics, the Error at any particular point can be assumed ~ **n(0, σ)**
Moreover, the Error at each point is independent of each other. Hence these different Errors are ***independent and identically distributed*** random variables.
Hence the Probability of the value of the Error at any particular point is **~n(0, σ).**

$$Y - h(X:\Theta) = \varepsilon \sim n(0 , \sigma)$$

Now, what we typically want to do in Gradient Descent is that we want to increase the Probability that our function returns Y for a corresponding X, parameterised by Theta.
Which means we want to Maximise the Conditional Probability of (Y | X: Θ)

$$Likelihood(\Theta) = Prob(Y \mid X:\Theta)$$
$$Likelihood(\Theta) \sim n(h(X:\Theta) , \sigma)$$

Iterating over the sampling dataset, we shall get that the Probabilities Multiply with each other and result ins something like this :

$$Likelihood(\Theta) = \prod Prob(Y \mid X:\Theta)$$

Taking a log:

$$\text{Log}(Likelihood(\Theta)) = K * \sum (Y - h(X:\Theta))^{**}2$$

This, as we see, is coming out to be similar to our cost function.
We want to maximise the probabilty that **h(X:Θ)** takes the value **Y**, hence we basically want to increase the **Likelihood(Θ)** and thereby the Log(**Likelihood(Θ)**). This turns out (on differentiation) turns out to be the exact same solution as for the minimisation of the MeanSquare Cost Function irrespective of **σ**.
Hence

Maximizing Likelihood ⇔ Minimising MeanSquare Error

Hence, the MeanSquare Error is the best Cost Function for Gradient Descent. And HyperParamaters like Number of Features, Dataset Size, Learning Rate are decided on the validation set and depend on a lot of values because these are all interdependent on each other.