ELL409 Assignment 1 Report

Kshitij Alwadhi (2019EE10577)

27th September 2021

1 Introduction

In this assignment we explored the concept of linear regression via polynomial curve fitting. For optimization, we use both, analytical method using Moore-Penrose pseudoinverse and Gradient Descent. We also implement different types of error metrics, batch gradient descent and regularization to avoid overfitting. We also implement cross-validation to judge the performance of the model using various hyper-parameters. At the end, we estimate the polynomial and also approximate the noise variance in the case of guassian noise.

2 Part 1A

2.1 Data Visualization

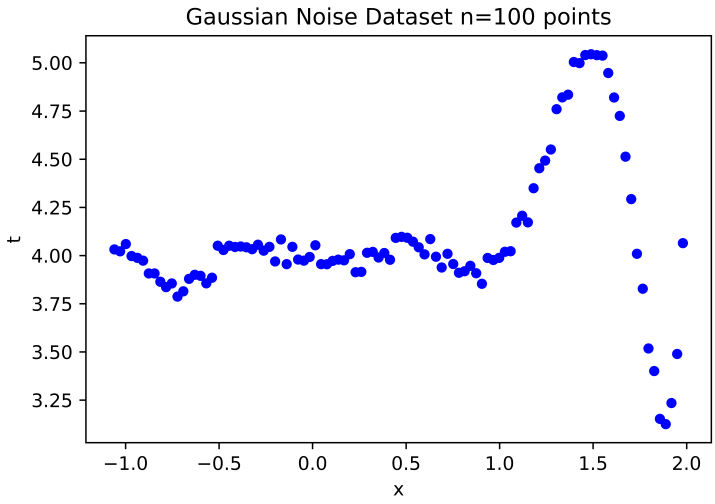

Figure 1: 100 data points

In this visualization, we can see a well structured data points and can imagine a curve going through them.

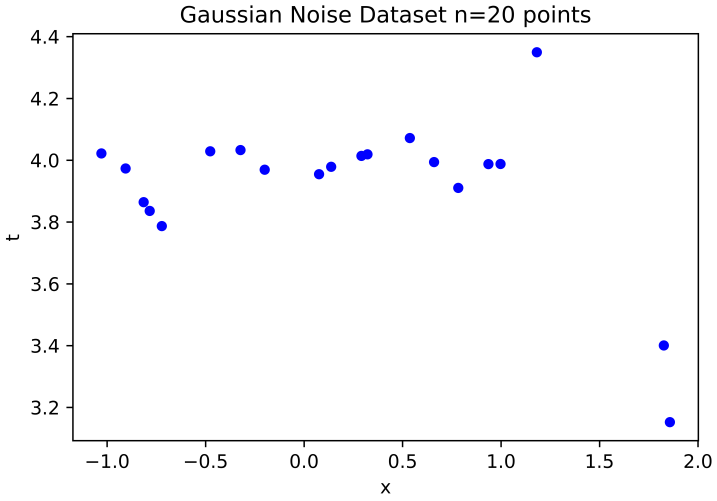

Figure 2: 20 data points

However, in this subset, the well structured pattern is lost so the model might have trouble in trying to figure out the polynomial.

2.2 Moore-Penrose Pseudoinverse optimization (For 20 points)

Running a loop over the values of m (degree of polynomial) from 3 to 12, we observe the following Training/Testing error vs m plot.

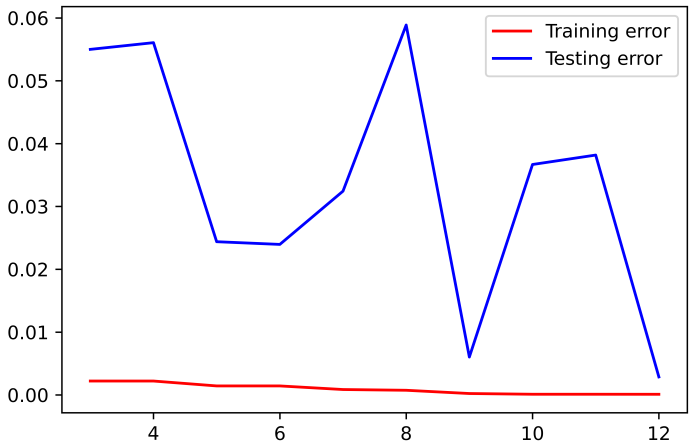

Figure 3: Training/Testing error vs Degree of polynomial

The data that we have in this case is very less, there are only 2 test data for us, hence it's very difficult to make out what the appropriate value of the degree should be. We see a minima of the testing error happening at m=9, hence we'll consider that as the best value of degree for the time being. Later, by looking at the plots, this assumption is confirmed.

2.2.1 Underfitting

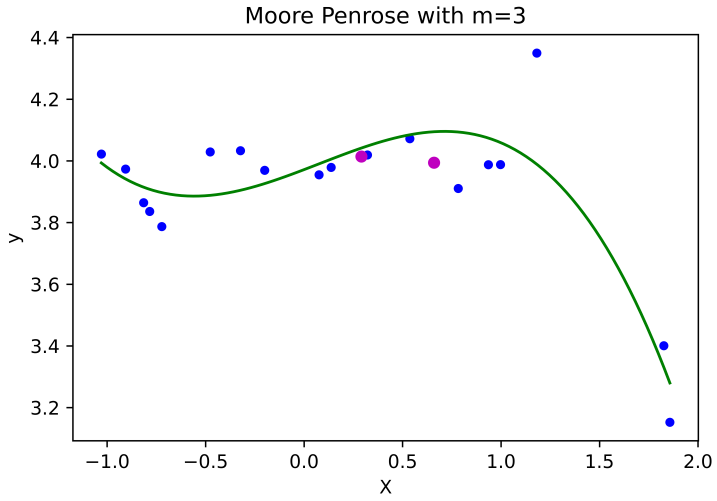

Figure 4: Degree = 3

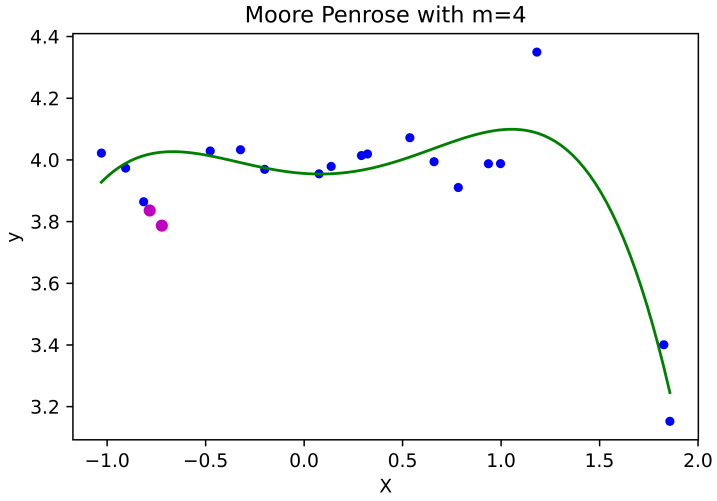

Figure 5: Degree = 4

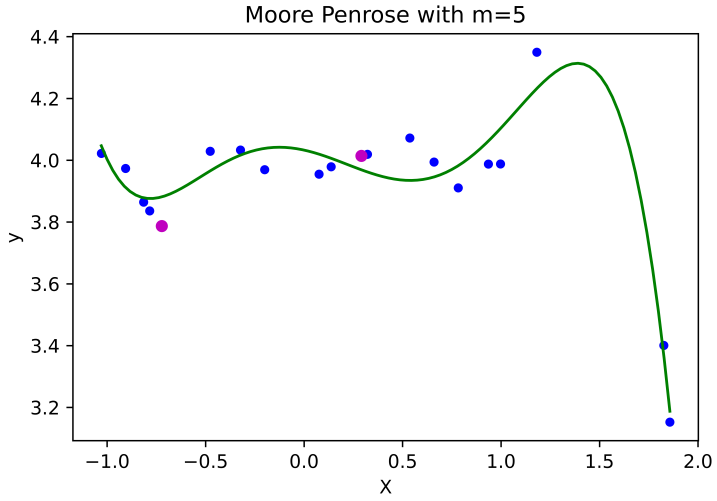

Figure 6: Degree = 5

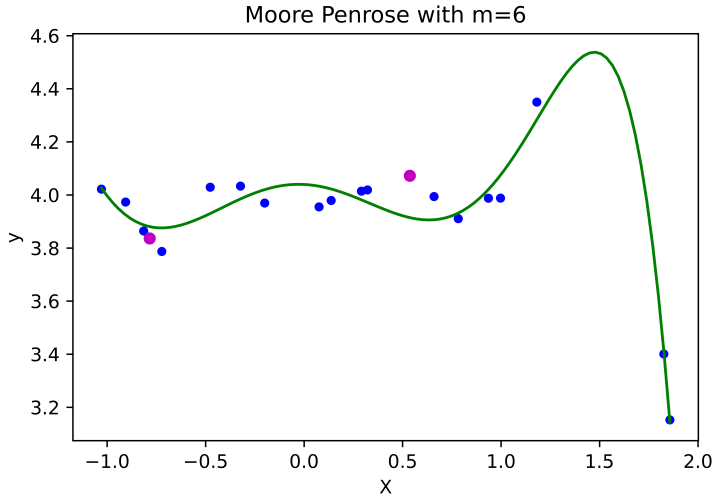

Figure 7: Degree = 6

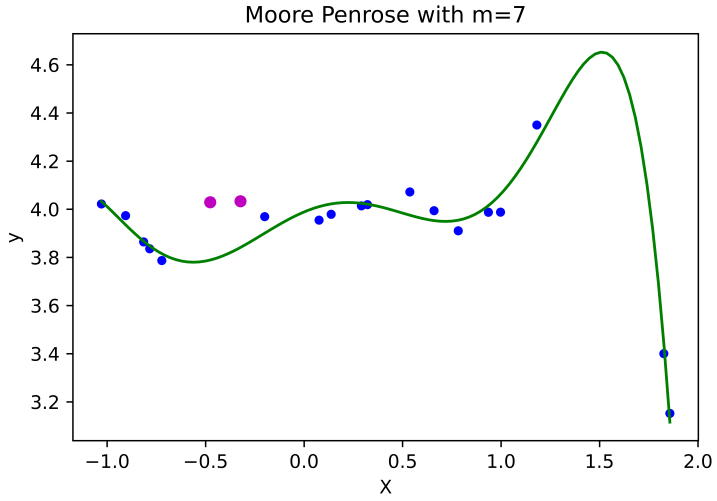

Figure 8: Degree = 7

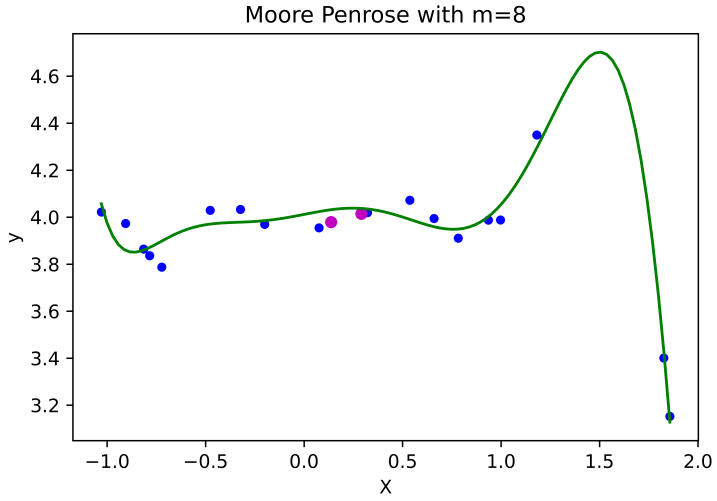

Figure 9: Degree = 8

2.2.2 Proper Fit

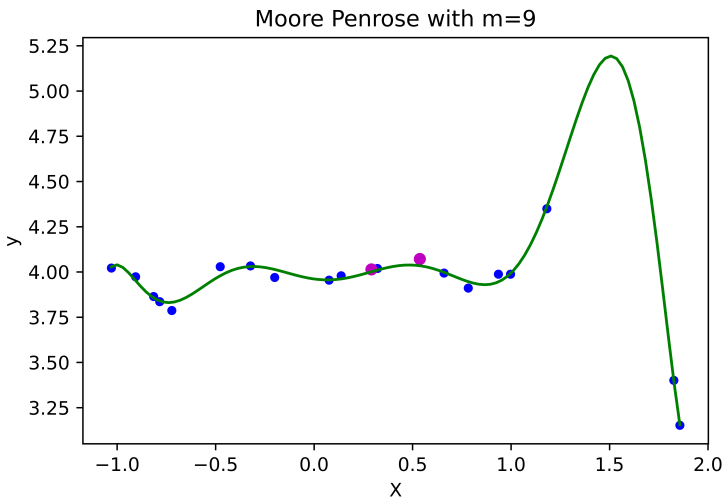

Figure 10: Degree = 9

2.2.3 Overfitting

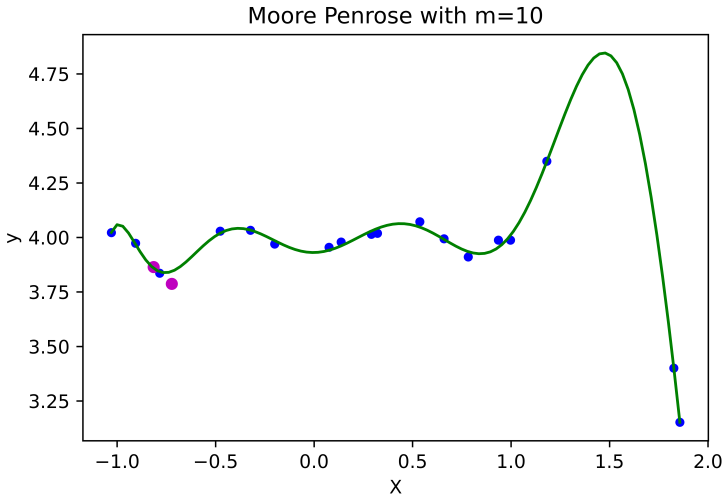

Figure 11: Degree = 10

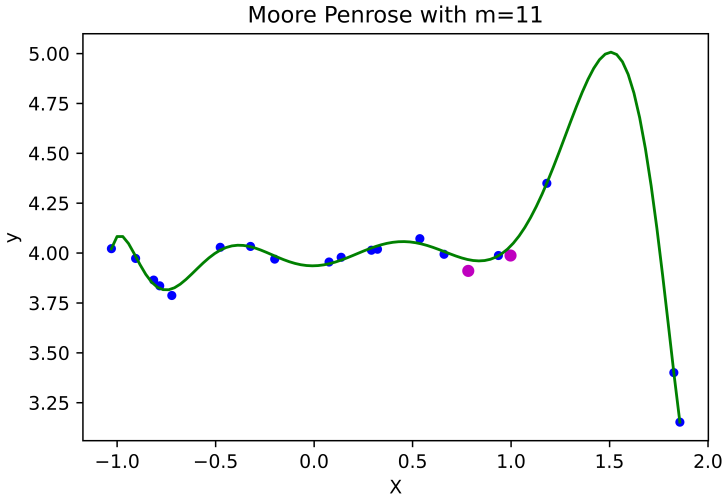

Figure 12: Degree = 11

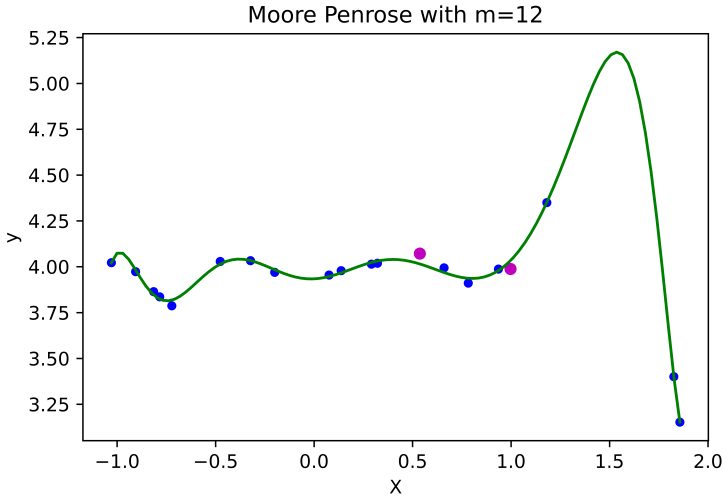

Figure 13: Degree = 12

2.2.4 Effect of regularization

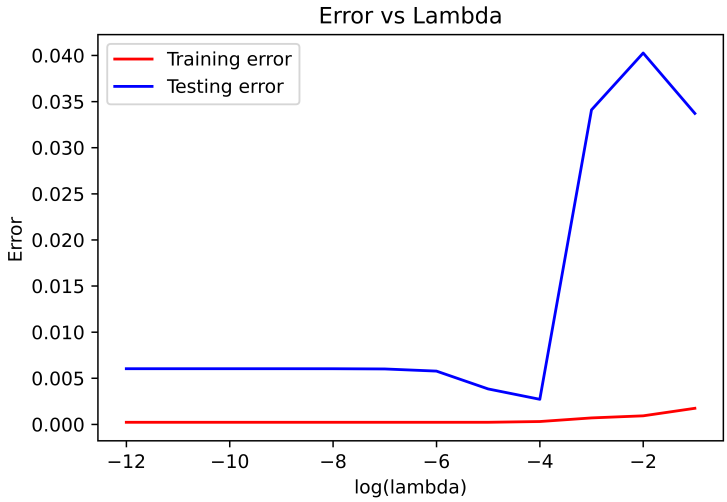

Figure 14: Error vs -log(lambda)

From the graph, we can see that lambda = $1e^{-4}$ is a good choice for this data and having a lambda greater than that will lead to underfitting.

2.2.5 Results

The weights predicted are [3.95792195, -0.12656421, 1.12464669, 1.26989771, -5.23816009, -1.57554556, 7.04337006, -0.51085981, -2.87296514, 0.91716426].

Therefore, our polynomial is: $3.95792195 - 0.12656421x + 1.12464669x^2 + 1.26989771x^3 - 5.23816009x^4 - 1.57554556x^5 + 7.04337006x^6 - 0.51085981x^7 - 2.87296514x^8 + 0.91716426x^9$.

2.3 Gradient Descent optimization (For 20 points)

2.3.1 Train/Test error vs Degree

We plot the variation of train/test error by change in the degree of the polynomial. Here we have fixed the batch size to be 64 and the regularization parameter to be 0.

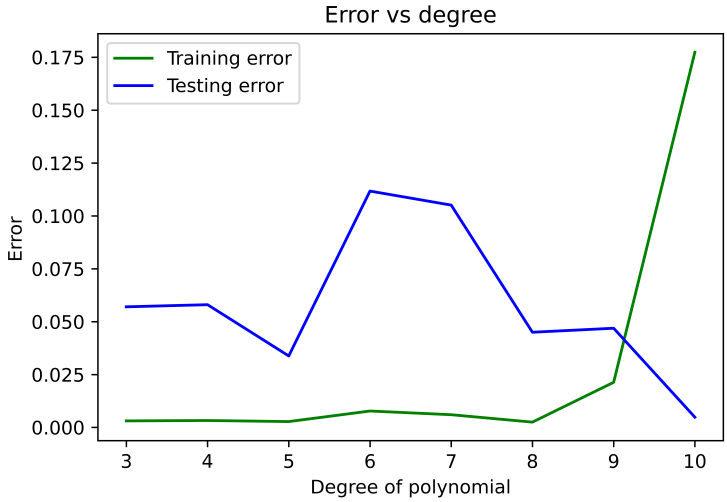

Figure 15: Train/Test error vs Degree of polynomial

We observe that both train and test error steadily decrease until m=8,9 after which they become relatively constant. This is in line with what we observed in the case of moore penrose pseudoinverse. For m<8, we observe underfitting and we find the sweet spot at m=9. Since the errors are not changing much beyond m=8, we settle with a simpler model (m=8).

Therefore, the degree of our polynomial is 8. (difficult to decide between 8 and 9)

2.3.2 Train/Test error vs Batch Size

We plot the variation of train/test error by change in the batch size of our learning algorithm. Here, we have fixed the degree of the polynomial to be 8 and the regularization parameter is 0.

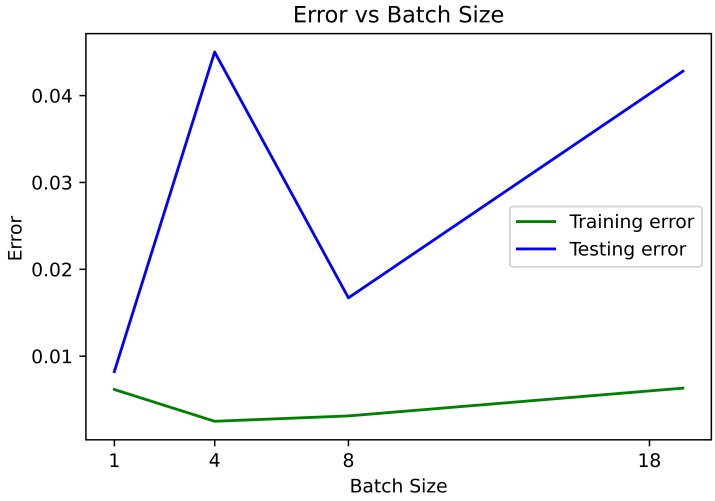

Figure 16: Train/Test error vs Batch Size

Here when the batch size is 1, that is basically stochastic gradient descent and when the batch size is 18, that is our full batch gradient descent. From the plot above, we observe that the best performance occurs when the batch size is 1 or 8. We use both of them as per our convenience in this experiment since 8 trains faster but 1 sometimes gives better results. Hence our batch size = 1,8.

2.3.3 Train/Test error vs Regularization constant

We plot the variation of train/test error by change in the regularization constant. Here, we have fixed the degree of the polynomial to be 8 and the batch size is 8 for faster training.

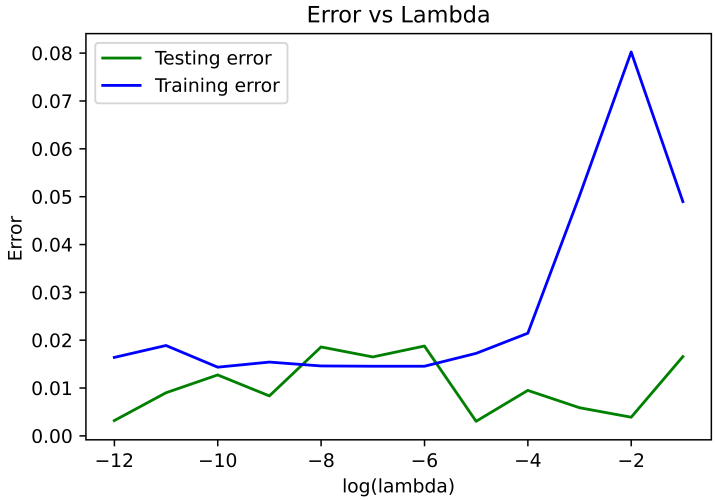

Figure 17: Train/Test error vs Regularization Constant

Here we observe a relatively constant training and testing error until $\lambda \leq 0.01$ after which the errors rise because of high penalty being imposed on the weights and hence the model tends to under fit. Therefore, to be on a safer side, we take the value of λ to be 10^{-6}.

2.3.4 Comparison with using different Errors

Apart from Mean Squared error, I also implemented the following two errors:

1. **Mean Absolute error**
2. **Huber Loss**

Analysis on them done in further sections when size of dataset is 100.

2.3.5 Learning rate optimization techniques

1. **Learning rate decay**: In this technique, the learning rate was updated after every iteration according to the following rule:
$$\alpha = \alpha * \exp \frac{-0.0015 * epoch}{NumIter}$$
where epoch is the index of the current iteration and NumIter is the total number of iterations.
2. **Normalized gradient**: This is more of a trick rather than a learning rate optimization. Here at the time of updation of weights, we divide the gradient by its norm so that we can get the direction the unit vector should take for decreasing the loss.

2.3.6 Results

The polynomial estimated from this experiment is: $y = 4.00769274 - 0.06324856x + 0.17222324x^2 + 0.71093897x^3 - 1.01716624x^4 - 1.06179476x^5 + 1.18786524x^6 + 0.40792198x^7 - 0.35595766x^8$
Shown below is the plot of this polynomial:

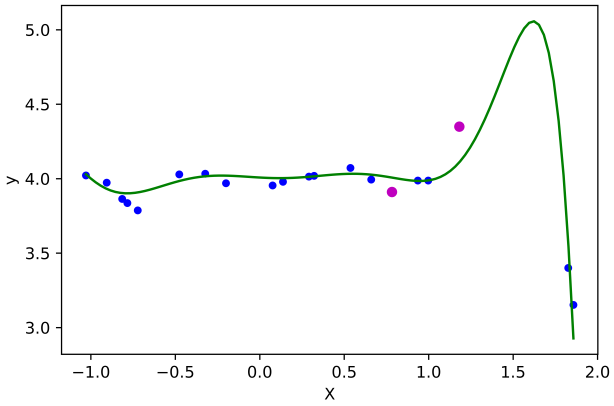

Figure 18: Plot of polynomial

BATCH SIZE = 1
LEARNING RATE = 0.003
REGULARIZATION $\lambda = 0.0001$
DEGREE = 8

2.4 Moore-Penrose Pseudoinverse optimization (For 100 points)

Running a loop over the values of m (degree of polynomial) from 3 to 12, we observe the following Training/Testing error vs m plot.

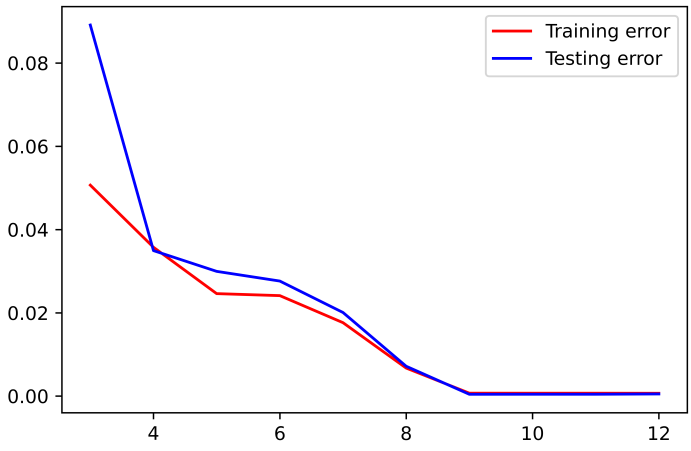

Figure 19: Training/Testing error vs Degree of polynomial

From this plot, we can see that the errors are high when m<9 (indicating underfitting) after which the training error keeps decreasing slightly but testing error starts rising from m≥12 indicating overfitting. The sweet spot occurs at m=9. This is also supported by the theorem which states that if two models are having similar performance, prefer the simpler model.

2.4.1 Underfitting

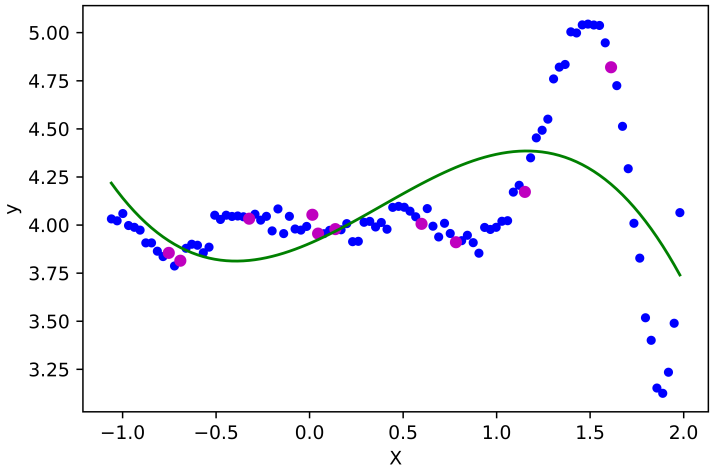

Figure 20: Degree = 3

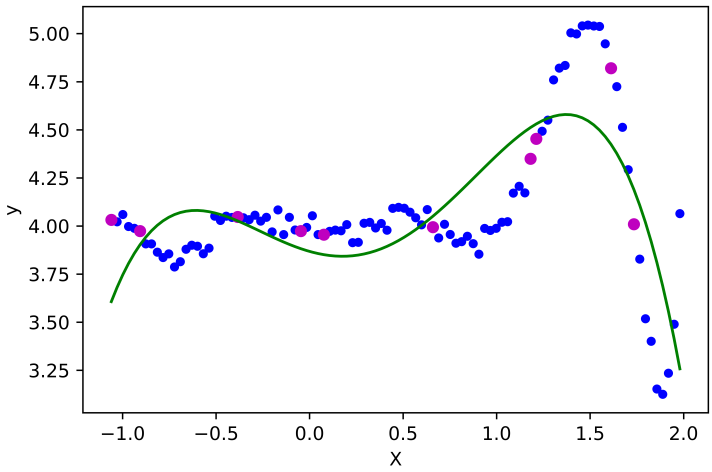

Figure 21: Degree = 4

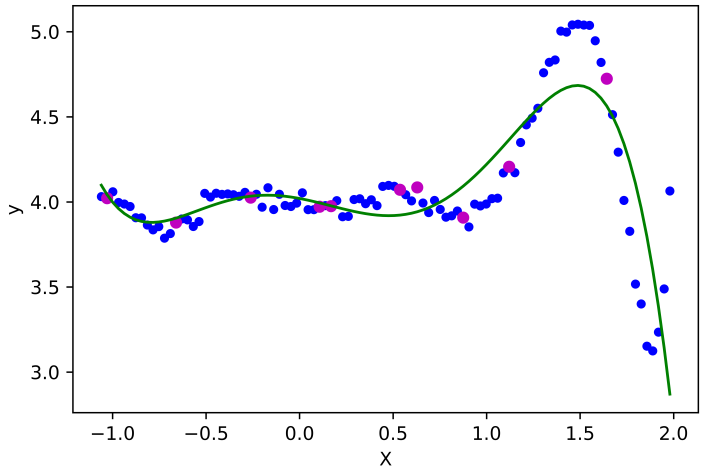

Figure 22: Degree = 5

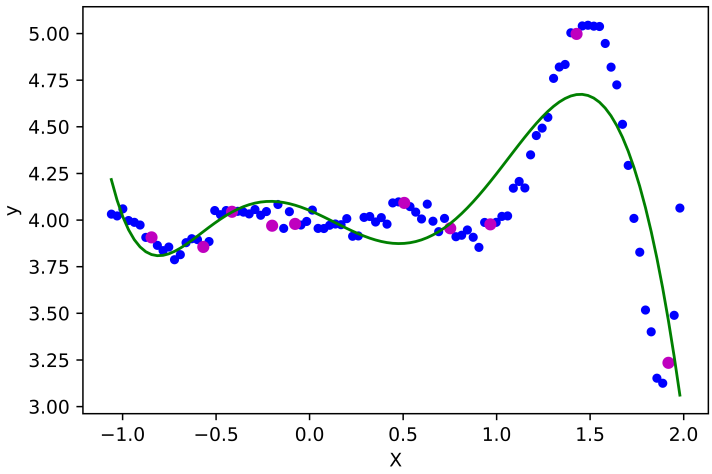

Figure 23: Degree = 6

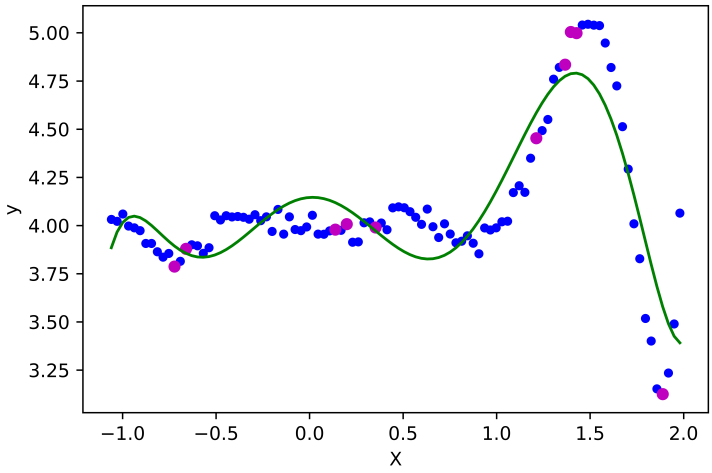

Figure 24: Degree = 7

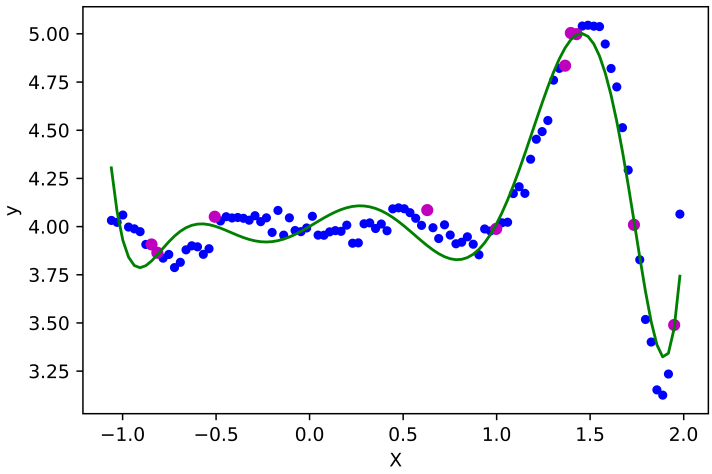

Figure 25: Degree = 8

2.4.2 Proper Fit

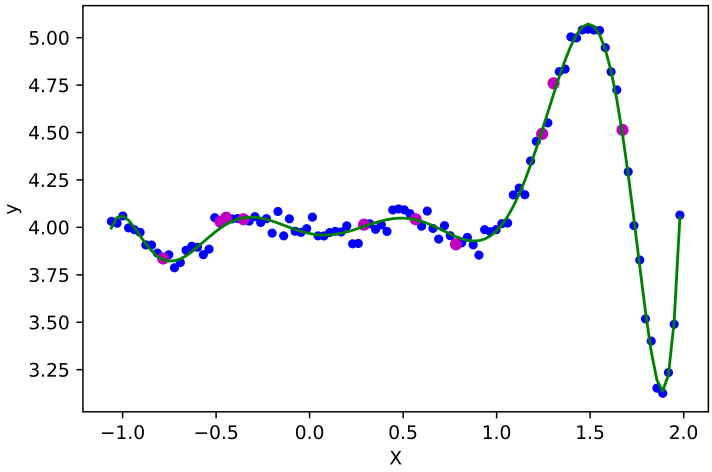

Figure 26: Degree = 9

2.4.3 Overfitting

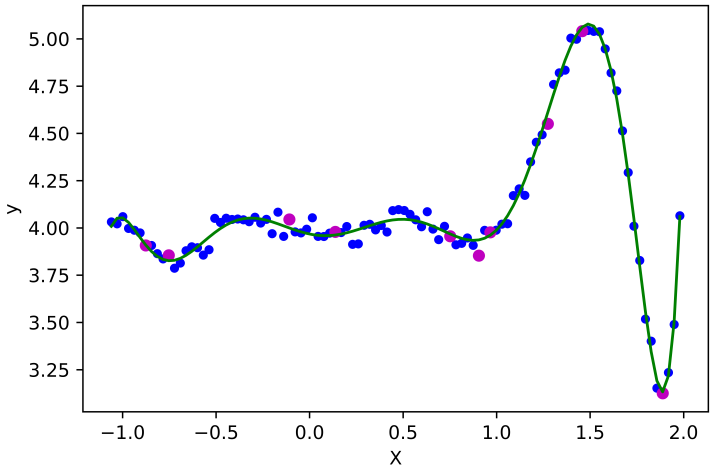

Figure 27: Degree = 10

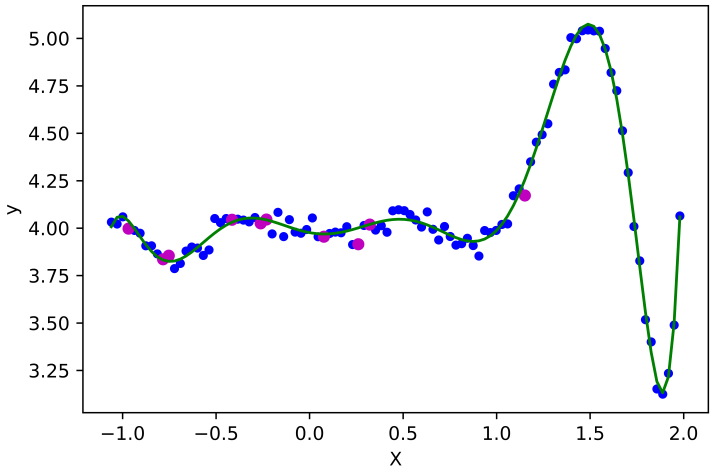

Figure 28: Degree = 11

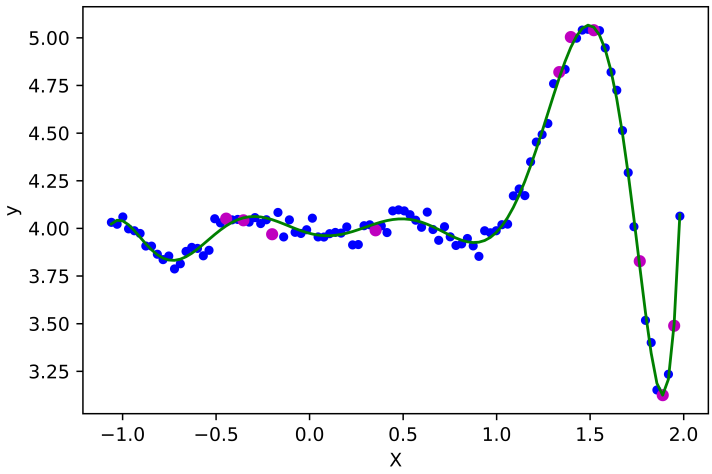

Figure 29: Degree = 12

2.4.4 Effect of regularization

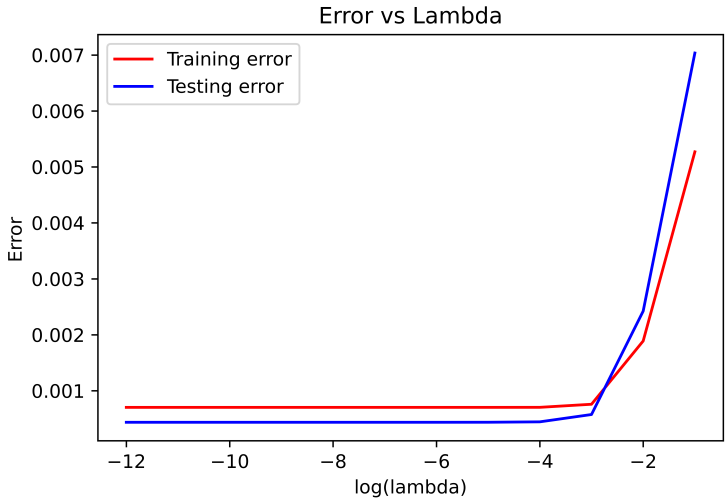

Figure 30: Error vs -log(lambda)

From the graph, we can see that lambda = $1e^{-4}$ is a good choice for this data and having a lambda greater than that will lead to underfitting.

2.4.5 Results

The weights predicted are [3.96888326, -0.20133548, 1.1389412, 1.53415068, -5.47669389, -1.77869129, 7.38728, -0.559866, -3.00083356, 0.96840988].

Therefore, our polynomial is: $3.96888326 - 0.20133548x + 1.1389412x^2 + 1.53415068x^3 - 5.47669389x^4 - 1.77869129x^5 + 7.38728x^6 - 0.559866x^7 - 3.00083356x^8 + 0.96840988x^9$.

2.5 Gradient Descent optimization (For 100 points)

2.5.1 Train/Test error vs Degree

We plot the variation of train/test error by change in the degree of the polynomial. Here we have fixed the batch size to be 64 and the regularization parameter to be 0.

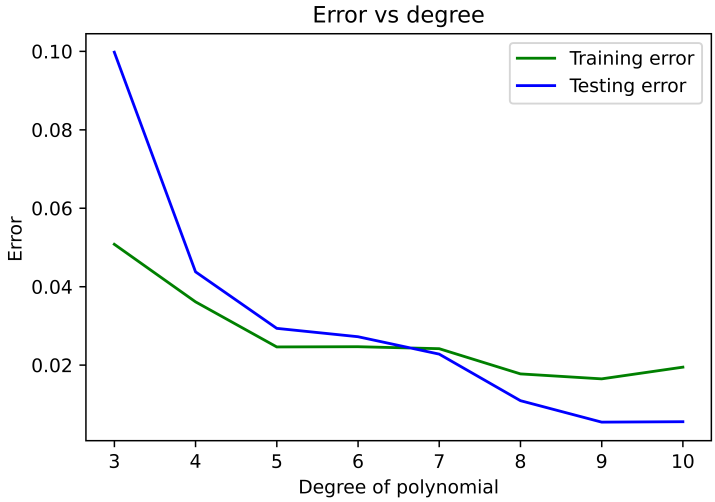

Figure 31: Train/Test error vs Degree of polynomial

We observe that both train and test error steadily decrease until m=8,9 after which they become relatively constant. This is in line with what we observed in the case of moore penrose pseudoinverse. For m<8, we observe underfitting and we find the sweet spot at m=9. Since the errors are not changing much beyond m=9, we settle with a simpler model (m=9).

Therefore, the degree of our polynomial is 9.

2.5.2 Train/Test error vs Batch Size

We plot the variation of train/test error by change in the batch size of our learning algorithm. Here, we have fixed the degree of the polynomial to be 9 and the regularization parameter is 0.

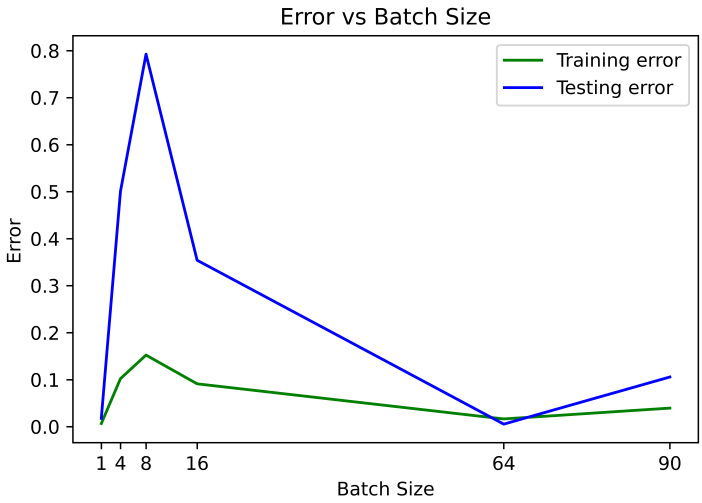

Figure 32: Train/Test error vs Batch Size

Here when the batch size is 1, that is basically stochastic gradient descent and when the batch size is 90, that is our full batch gradient descent. From the plot above, we observe that the best performance occurs when the batch size is 1 or 64. We use both of them as per our convenience in this experiment since 64 trains faster

but 1 sometimes gives better results. Hence our batch size = 1,64. Generally, if batch size is increased for a fixed number of total iterations, the time taken for the program will increase.

2.5.3 Train/Test error vs Regularization constant

We plot the variation of train/test error by change in the regularization constant. Here, we have fixed the degree of the polynomial to be 9 and the batch size is 64 for faster training.

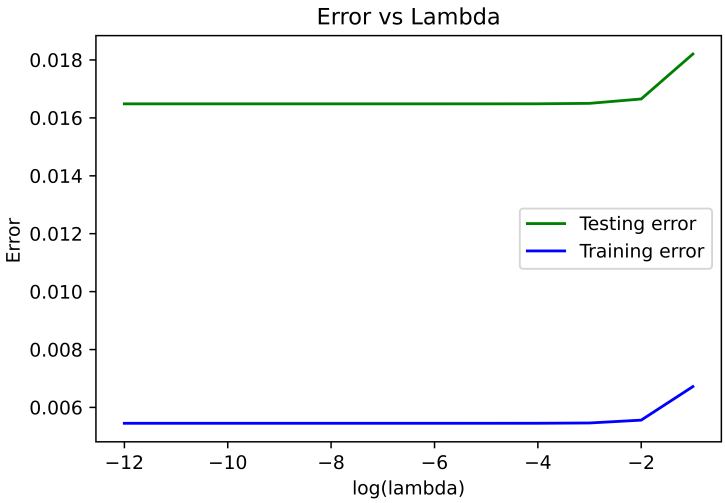

Figure 33: Train/Test error vs Regularization constant

Here we observe a relatively constant training and testing error until $\lambda \leq 0.01$ after which the errors rise because of high penalty being imposed on the weights and hence the model tends to under fit. Therefore, to be on a safer side, we take the value of λ to be 10^{-6}.

2.5.4 Comparison with using different Errors

The following are the error functions I implemented (used their respective gradients for updating the weights):

1. **Mean Squared error:** Defined as following:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_{pred})^2$$

BATCH SIZE = 4

LEARNING RATE = 0.003

NUMBER OF ITERATIONS = 1000000

REGULARIZATION $\lambda = 0$

DEGREE = 9

After fitting, the following is the plot we get:

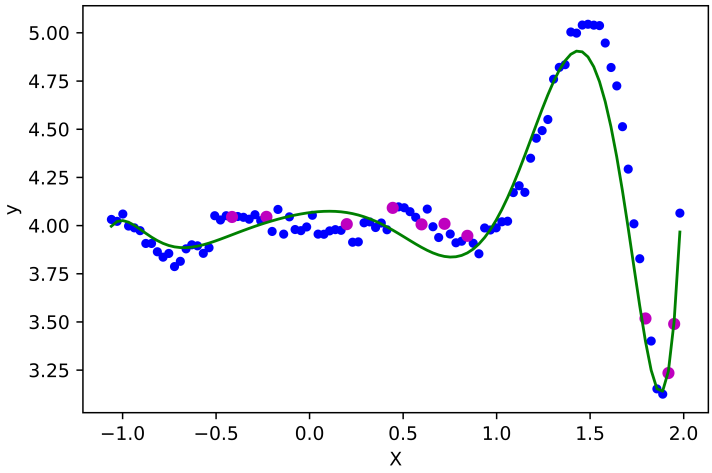

Figure 34: Gradient Descent with MSE

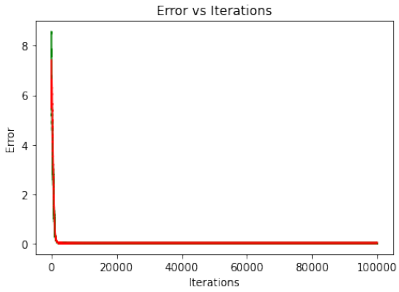

Figure 35: Loss vs iterations (Green: Train, Red: Test

The loss starts off high and falls very fast in the first couple of iterations. After that, it falls very slowly. The MSE error I got for this was: 0.007716105577904854.

2. **Mean Absolute error:** Defined as following:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - y_{pred}|$$

BATCH SIZE = 64
LEARNING RATE = 0.003
NUMBER OF ITERATIONS = 1000000
REGULARIZATION $\lambda = 0$
DEGREE = 9
After fitting, the following is the plot we get:

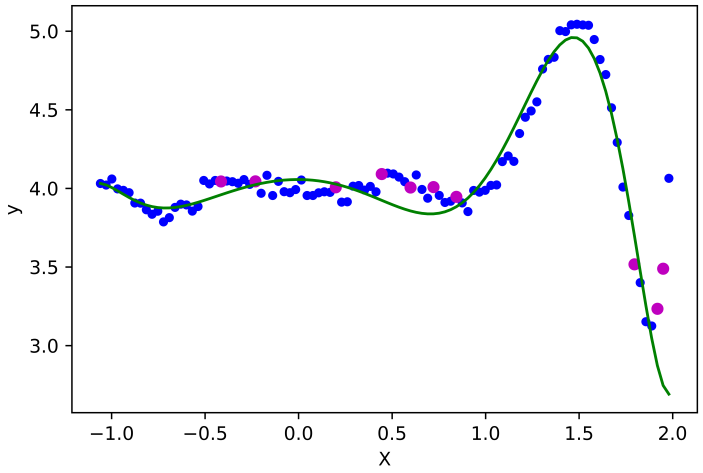

Figure 36: Gradient Descent with MAE

The MSE error I got for this was: 0.014430131610707964.

3. **Huber Loss:** Defined as following:

$$L_H = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

where, a $= y_{pred} - y_i$,

and $\delta = 0.5$ (Hyperparameter).

BATCH SIZE = 64

LEARNING RATE = 0.003

NUMBER OF ITERATIONS = 100000

REGULARIZATION $\lambda = 0$

DEGREE = 9

After fitting, the following is the plot we get:

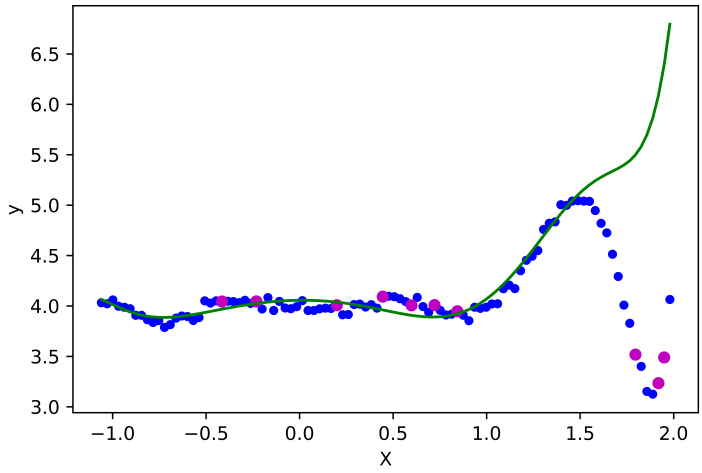

Figure 37: Gradient Descent with HUBER

The MSE error I got for this was: 0.1866222709752176.

Clearly, for the given data and for the parameters which we tuned, Mean squared error was the best option.

2.5.5 Learning rate optimization techniques

1. **Learning rate decay**: In this technique, the learning rate was updated after every iteration according to the following rule:
$$\alpha = \alpha * \exp \frac{-0.0015 * epoch}{NumIter}$$
where epoch is the index of the current iteration and NumIter is the total number of iterations.

2. **Normalized gradient**: This is more of a trick rather than a learning rate optimization. Here at the time of updation of weights, we divide the gradient by its norm so that we can get the direction the unit vector should take for decreasing the loss.

2.5.6 Results

The polynomial estimated from this experiment is: $y = 3.96832055 - 0.13573756x + 1.07340517x^2 + 1.08750076x^3 - 5.04570011x^4 - 1.15167529x^5 + 6.66411332x^6 - 0.7004318x^7 - 2.64154578x^8 + 0.86910988x^9$
Shown below is the plot of this polynomial:

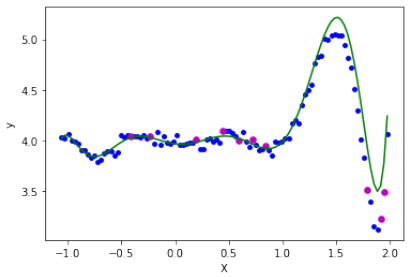

Figure 38: Plot of polynomial

BATCH SIZE = 1

LEARNING RATE = 0.003
REGULARIZATION $\lambda = 0.0001$
DEGREE = 9

2.6 Difference between the two cases (Number of data points)

2.6.1 Fitting lower degree polynomials

One major difference that we see is that in the case of 20 points data, even smaller degree polynomials seem to give a very decent fit. Had we not seen the picture of the full data, it would have been very difficult to predict what the degree should be.

2.6.2 Sensitivity to test train split

Even our choice of train test split can make a huge difference when we are considering only 20 points. The error vs degree plot would change entirely had we not performed cross validation to calculate the error.

2.6.3 Sensitivity of the minima to the error

When performing batch gradient descent, we observe that having more number of points lead to faster convergence. Hence, the convergence speed is lower in the case of 20 data points. Thus requiring us to run extra iterations or tune the learning rate.

2.7 Final Estimate of the Polynomial

Our final estimate of the polynomial (best results using the analytical method as evident from the graphs using all the data) is:

$y = 3.96888326 - 0.20133548x + 1.1389412x^2 + 1.53415068x^3 - 5.47669389x^4 - 1.77869129x^5 + 7.38728x^6 - 0.559866x^7 - 3.00083356x^8 + 0.96840988x^9$.

The variance of the Gaussian noise is: 0.0013441315056887191.

3 Part 1B

In this part, we are given a dataset with a Non-Gaussian noise. We perform a similar analysis as in part 1A using moore penrose pseudoinverse. We find an optimal value of degree (m=11) and regularization parameter $= 10^{-4}$.

3.1 Data Visualization

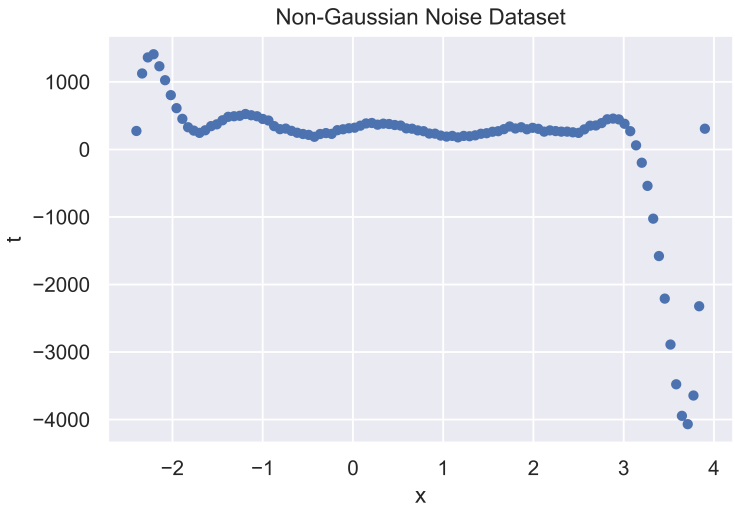

Figure 39: Non-Gaussian Noise dataset

3.2 Noise Estimation

The noise in our dataset can be represented as:

$$Data = UnderlyingPolynomial + Noise$$

$$Noise = Data - UnderlyingPolynomial$$

Now using the Moore-Penrose Pseudoinverse technique we estimate the underlying polynomial so that we can make further observations. It comes out to be a 10 degree polynomial with the weights as: [329.7513736, 349.45711345, -299.31282937, -954.43631167, 575.88250596, 591.01392295, -361.11524098, -116.54322939, 88.95651221, 1.32873283, -7.31015015, 1.00032137].

The following is the graph that we get after fitting the polynomial

Figure 40: Non-Gaussian Noise polynomial

Now to estimate the type of noise, we create an array containing the difference between the predicted values and the true values given to us in the dataset.

$$noise = t - preds$$

Next, we plot a histogram for this noise array so that we can see its distribution.

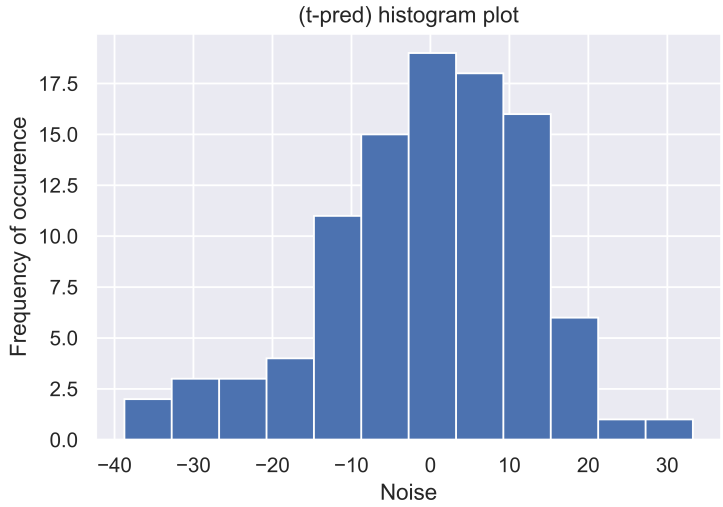

Figure 41: Non-Gaussian Noise Distribution

Moreover, from the array, we also calculate the noise and the standard deviation of the noise distribution, which comes out to be:

Mean = 0.000335534 $\approx$ 0.

Standard Deviation = 13.5487918.

Minimum noise = -38.7547405461122.

Maximum noise = 37.186325280103574.

First of all, we are dealing with a 0 mean noise distribution. Secondly, we are given that this is not a Gaussian Distribution, so we can leave the normal distribution out of the picture. Now, we move on to the

histogram that we got.

The shape of the curve is similar to that of a Beta function with $\alpha = 8$ and $\beta = 4$ as illustrated below.

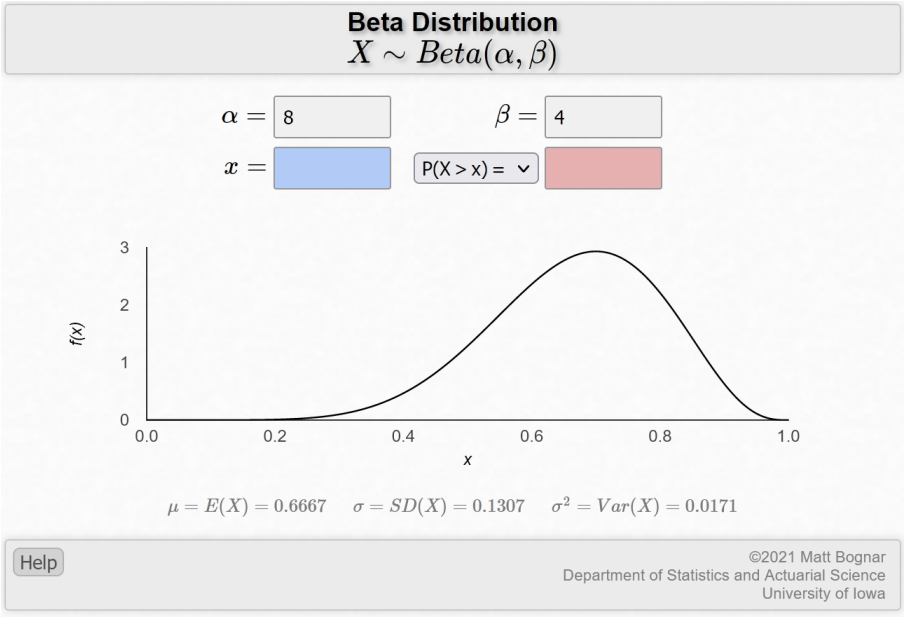

Figure 42: Beta Distribution

One difference that we observe is that the distribution we have got has a mean of 0, so we will have to account for that. So what we can do this, map the X axis of this beta distribution to the histogram that we got so as to get a reasonable estimate of the error distribution. We could have also modelled Poisson distribution for this task.

Another approach that we could have tried is, instead of directly taking the difference between pred and true value, take the mod of it(absolute value). The following is the distribution graph that we get from it.

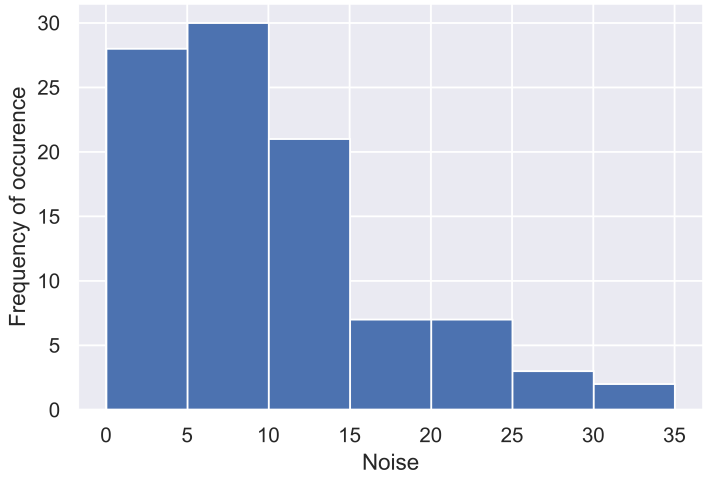

Figure 43: Non-Gaussian Noise Distribution

From this, a possible noise distribution would be an exponential function with a relatively smaller value of lambda. Moreover, since we also have errors as negative sometimes, we can model the noise distribution as the

product of two probability distributions, i.e. the exponential distribution function and a Bernouli distribution with p = 0.5 for deciding the sign of the noise (positive or negative).

One more approach that I tried was using the Fitter library built for python. It has various distributions built into it and can fit the data given into them and provide the best results. The following are the results from the library:

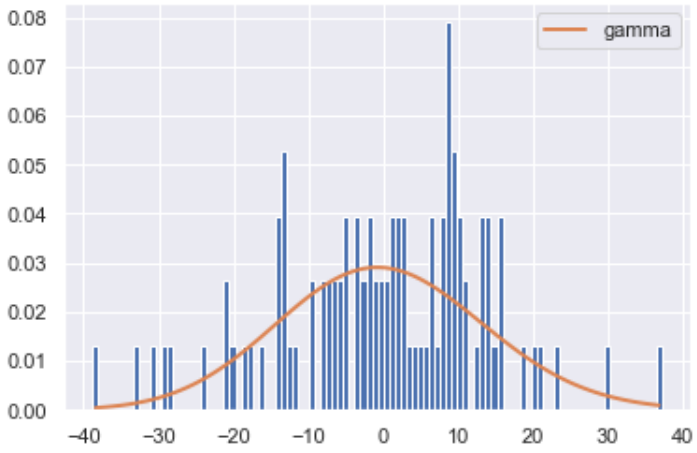

Figure 44: Fit for (pred - true value)

When fitting the (pred - true value) array into the distributions, it got the best results by fitting the Gamma distribution as shown in the plot above.

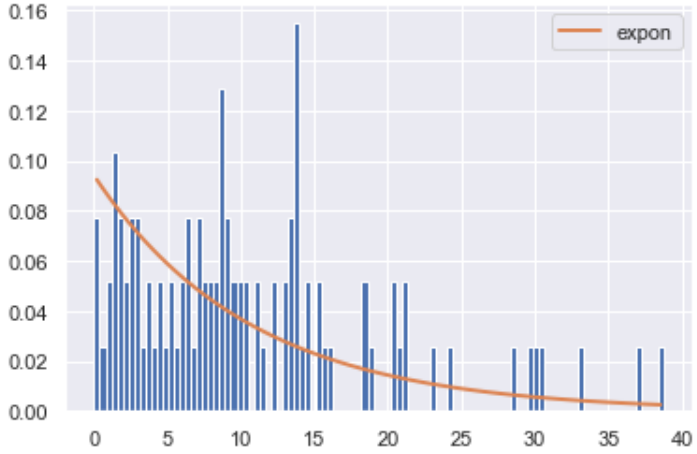

Figure 45: Exponential Fit for abs(pred - true value)

Next, when the absolute value was used along with the exponential distribution, the results were as shown above.

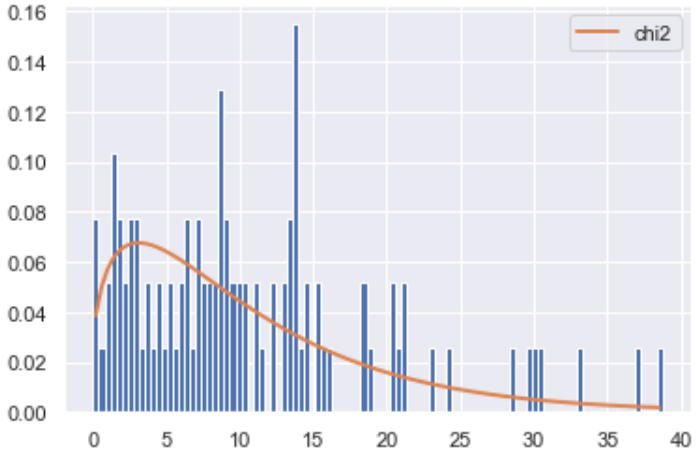

Figure 46: Chi-2 Fit for abs(pred - true value)

Moreover, the best results when fitting the absolute values was found using the chi-squared probability distribution as shown in the plot above.

4 Part 2

In this part of the assignment, we were given a dataset containing two columns, the date and the corresponding value. Firstly, we split the date into two separate features, the month and the year (ignoring date since it's 1 for every data entry).

4.1 Data Visualization

To visualize the data, we plot the values of a particular year (say 2005) for the corresponding months on the x-axis. We obtain the following plot:

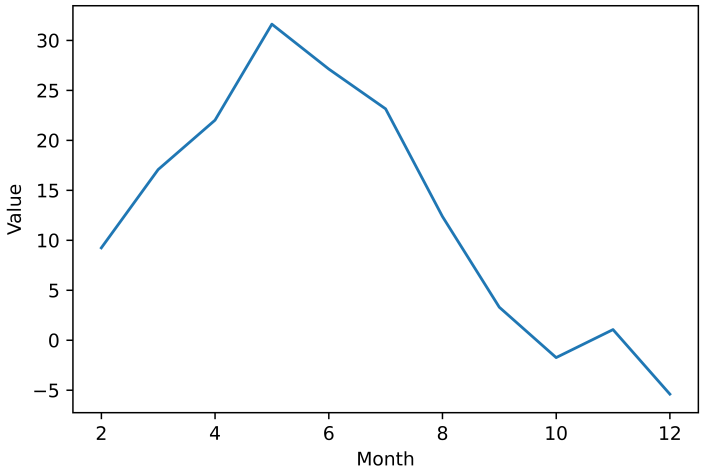

Figure 47: Values vs months plot for year 2005

One clear observation that we can make from this data is that the these are the temperatures of some particular location in Northern India since values increase during the months of summer then go down during winters. This pattern is consistent across all the years.

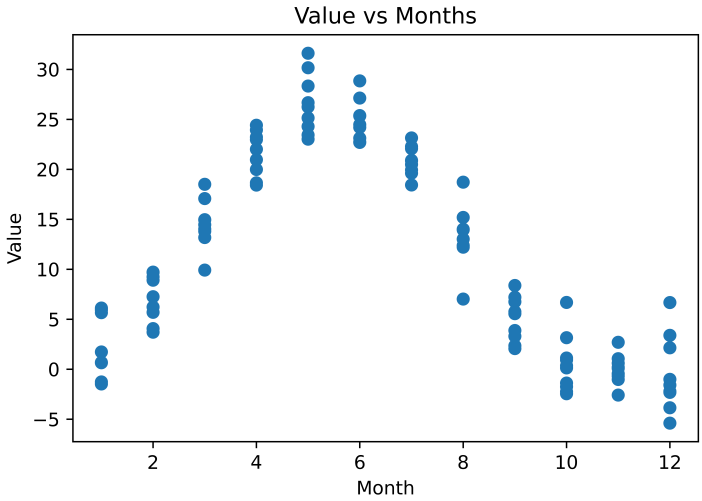

Figure 48: Scatter plot

4.2 Feature Selection

For selecting the features, we split the date into two separate entities, the month and the year (ignoring date since it's 1 for every data entry). Next, since we will be using polynomial regression for this problem, we need to create a design matrix for these features. Now in the design matrix, we need to decide the degree for both of these features. We'll call them degreeM, degreeY indicating the degree of month feature and degree of year feature respectively.

4.3 Model Training

For training the polynomial regression model, we use the Moore-penrose pseudoinverse optimization. The training data given to us (110 points) was used for training in a 10-fold cross validation setting where 99 points were being used for training and the rest 11 were used for test set. Moreover, since we dont know the value of degreeM and degreeY, we perform a grid search over a range of (0,15) for finding the appropriate values. The grid search also incorporated a wide range of regularization values.

The following plots were obtained:

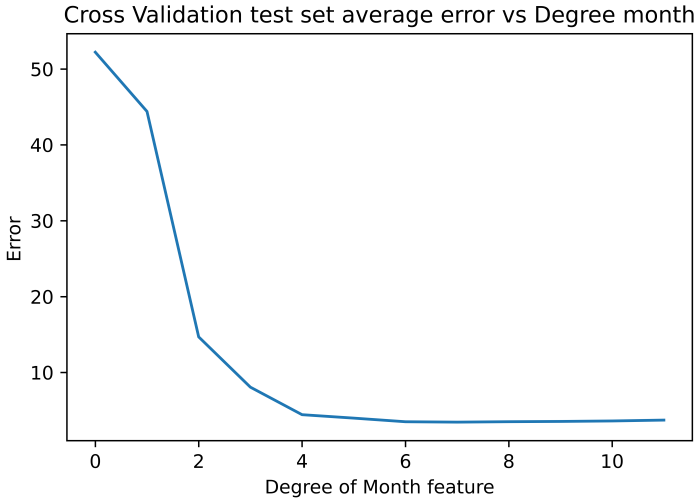

Figure 49: Cross Validation test set average error vs Degree month

In the above plot, we set the degree of year feature to be 0 and the λ to be 0 and plotted the change in error with variation in the degree of month. We see that the lowest error occurs when the degree is 6 and it becomes stable afterwards. Hence, taking the simplest model with degree of month = 6.

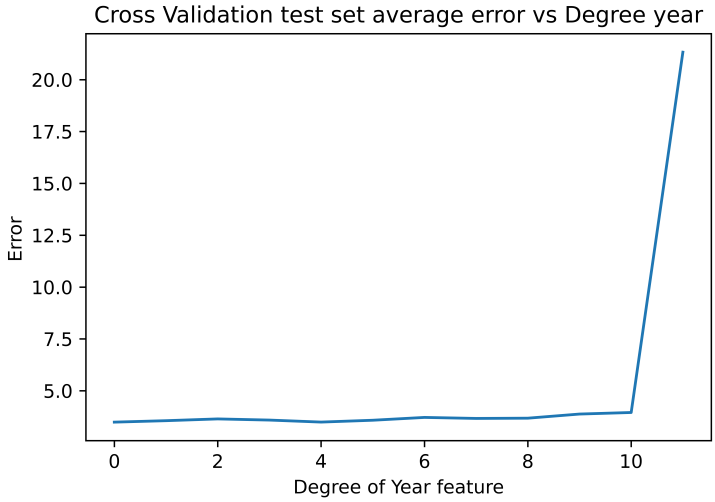

Figure 50: Cross Validation test set average error vs Degree year

In the above plot, we set the degree of month feature to be 6 and the λ to be 0 and plotted the change in error with variation in the degree of year. We see that the lowest error occurs when the degree is 0. It increases very slightly until 10, after which it explodes. Hence, taking the simplest model with degree of year = 0.

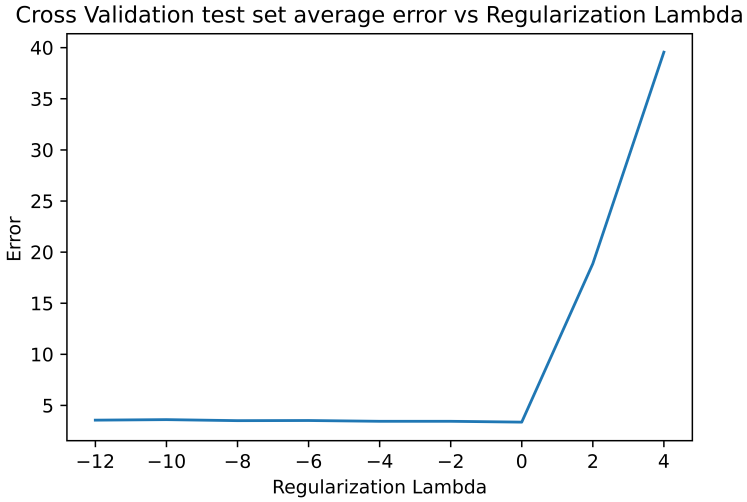

Figure 51: Cross Validation test set average error vs Regularization λ

In the above plot, we set the degree of month feature to be 6 and the degree of years feature to be 0 and plotted the change in error with variation in the regularization parameter. We see that error is almost same for all values of $\lambda \leq 1$ beyond which, the error shoots up as the weights are being forced to take small values.

At the end of this grid search, the following parameters were set for training the model:

degreeM = 6

degreeY = 0

lambda = 0.35

Surprisingly, the best fit was found when the degree of year feature was set to be 0.

Error is: 1.0651125264570256

Weights are: [5.39774912e-01 8.86021830e-01 4.76440027e-02 1.09439008e+00 -3.05973091e-01 2.79874802e-02 -8.46985034e-04]

4.4 Evaluation

Using, the following hyperparameters,
degreeM = 6
degreeY = 0
lambda = 0.35
K-Fold Cross validation = 10
We got the following prediction:

id	value
05-01-2010	26.00127005
04-01-2009	21.73032962
09-01-2013	5.221146117
01-01-2006	2.599273239
02-01-2007	7.248245718
08-01-2012	12.87456752
06-01-2014	25.63189133
03-01-2008	14.53288773
12-01-2004	-0.433679437
07-01-2011	20.65845357

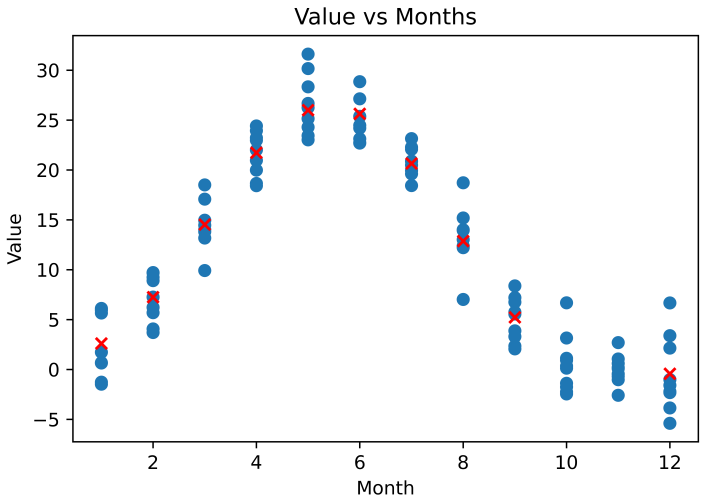

Figure 52: Predictions Scatter plot

On Kaggle, this got me a score of 5.33459.

However, to my surprise, one of the submissions that I tried out had the the average value over the dataset for the corresponding month for which the prediction is required, and this performed better than my regression model. It got a score of 5.15135 on Kaggle. I also tried modelling my predictions as a weighted mean of the prediction by regression model and the average values as following:

$$y_f = \alpha * y_{reg} + (1 - \alpha) * y_{avg}$$

However, the best score came with having the value of $\alpha = 0$.

4.5 Learnings from this task

1. Usage of K-fold cross validation is very helpful especially in the case of limited dataset since it eliminates the "luck" factor while making predictions on the test set.

2. Initially, I was manually trying out various hyperparameters to see which performed the best. Later I switched to performing a grid search for tuning the hyperparameters which reduced my job significantly.

3. More feature modeling was needed for this task. A model which took into account the previous months' temperature would have worked better since this is a type of temporal forecasting.