ELL409

Assignment 1

Vansh Gupta

2019EE10143

Part 1A

- ### Using only 20 data points

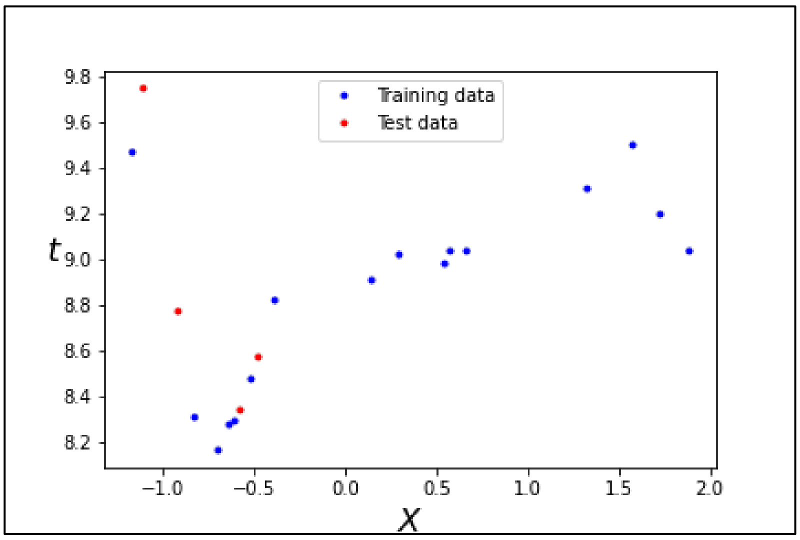


Fig 1. Randomly sampled 20 data points

- ### Using Moore-Penrose pseudoinverse:

I run my code for different values of λ (regularization constant) and epochs (number of iterations). [Fig 2.]

After that, I observed that the number of epochs have no effect on the losses, which is also expected since our dataset is very small for the model to have to learn the same data point again. Moreover, this method is not exactly learning but presenting with a solution. Thus, I iterate over different λ. We can also see that while the training error has an elbow point at m=5 (m is the highest degree of polynomial), the test set is truly reduced for m=9. Thus, for further iterations, I will fix m at 9. [Fig 3]. Here we get to witness overfitting and underfitting as the model gives good training set performance for higher degree but poor test set accuracy and poor training and test set accuracy for lower values of M.

Now, we see that that there is not much change for a range of values for λ. Even a value of 0 seems to be working fine suggesting that there is no overfitting that needs to be mitigated. This is possibly because with only 16 training points, it is like data is missing at lots of places, and since there is also noise in our data, the model is unable to exactly learn the curve or even overfit it.

A final plot for λ = 0, number of epochs = 1, over different values of m is shown in Fig. 4

Therefore, the best guess of the polynomial, for m = 9 and λ = 0 is:

$$8.99654324 - 0.32089103\,\boldsymbol{X} - 0.30898259\,\boldsymbol{X}^2 + 4.58312801\,\boldsymbol{X}^3 - 3.91859305\,\boldsymbol{X}^4 - 6.12614492\,\boldsymbol{X}^5 + 7.53175079\,\boldsymbol{X}^6 + 0.88516857\boldsymbol{X}^7 - 3.20999694\,\boldsymbol{X}^8 + 0.88288049\,\boldsymbol{X}^9$$

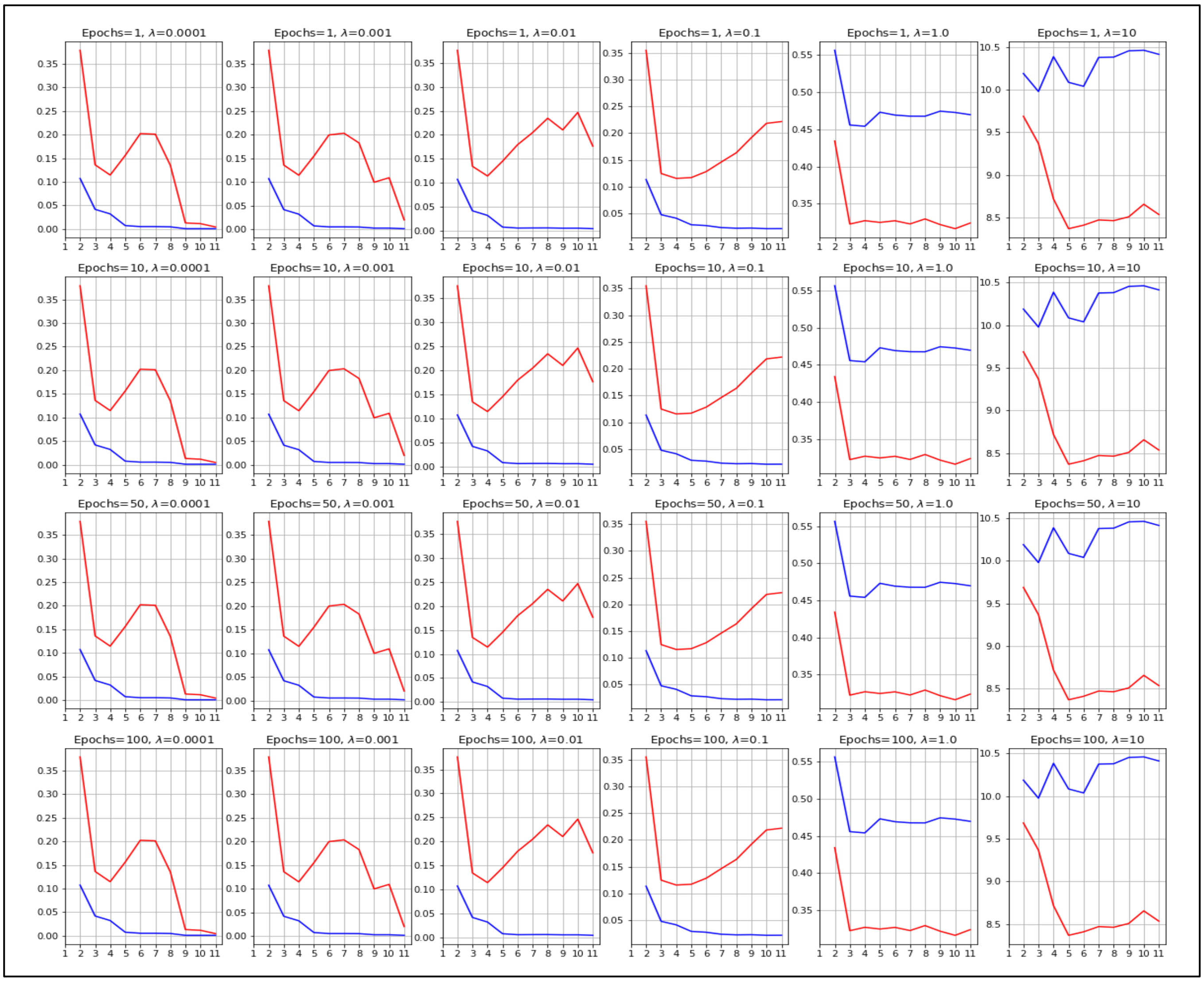


Fig 2. The red line depicts the test error and the blue line shows the training error

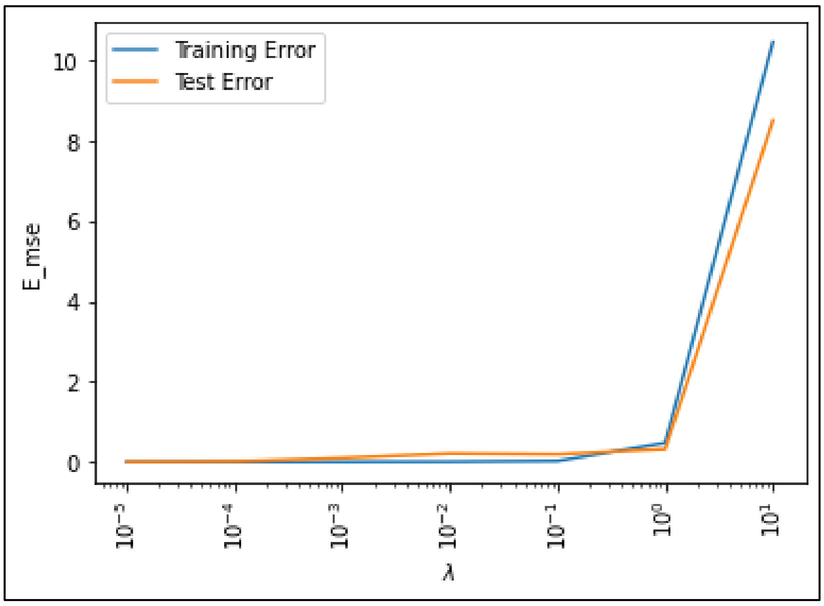


Fig 3.

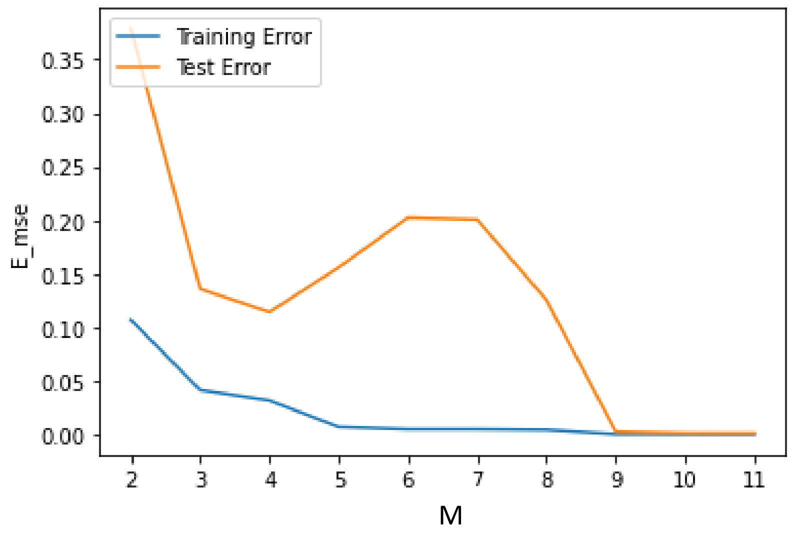


Fig 4.

• Using Gradient Descent:

I followed the exact same procedure as listed for the Moore-Penrose pseudoinverse.

The inference for λ was absolute and not related to the method, therefore for this set of experiments, I keep λ to be 0. I also keep my epochs i.e., the number of iterations to be at a fixed value of 1500. This is because the data is small, and for gradient descent which does not produce a closed form solution, needs to go over the data multiple times to not succumb to underfitting

Thus, here my main focus is on the maximum degree of the polynomial (which should be the same), the learning rate η and the size of our mini-batch.

First, for the learning rate, after trying various different values for η, I came to the conclusion that there is no single good value for the learning rate and that it needs to change with time. Thus, once again after extensive experimentation, I found the value of $\frac{0.3}{i+1.7}$ to work best for the given data. Here, "i" is the starting index for a certain mini-batch.

For the order of polynomial, I take number of epochs and learning rate value as described above and do mini-batch gradient descent on it for a batch size of 10. The results are in Fig. 5.

Note: For all models of gradient descent, I have used a min-max normalization followed by subtracting each dimension by its mean for faster and better convergence. The same normalizing parameters (Minimum, maximum, mean of min-max normalized training set) is used to normalize the test set. However, since these weights will not match for unnormalized data, I have thus removed this from the code submitted for auto-grading, and naturally the error increased by a lot, as the curve is most likely not fitting. In a real-world setting, I can store the parameters of normalization and use those for new data. Consequently, I change my learning rate and number of epochs as well.

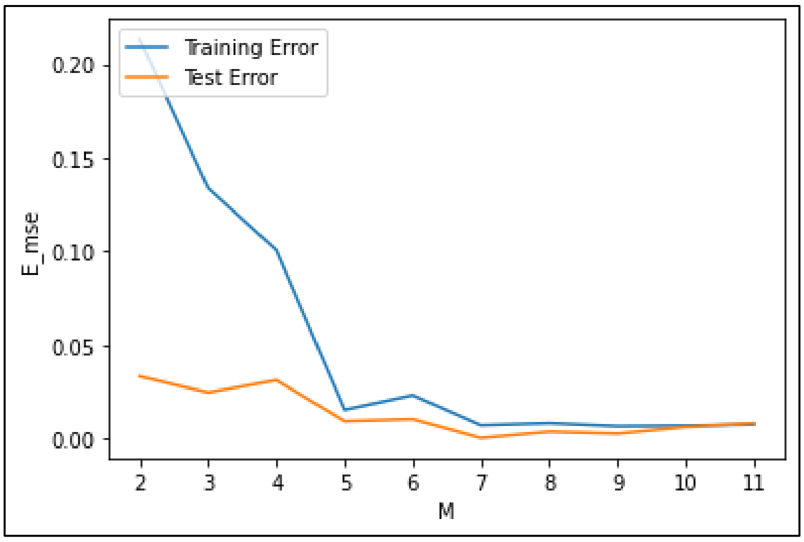


Fig. 6: Blue line is training error while the orange line is test error

This is not something very good, but keeping in mind that there are only 20 data points, we will have to work with it. Highest degree 7 seems good enough for our gradient descent model.

Using 7 as our M, we know vary batch size from 1 to 16 (Size of training set)

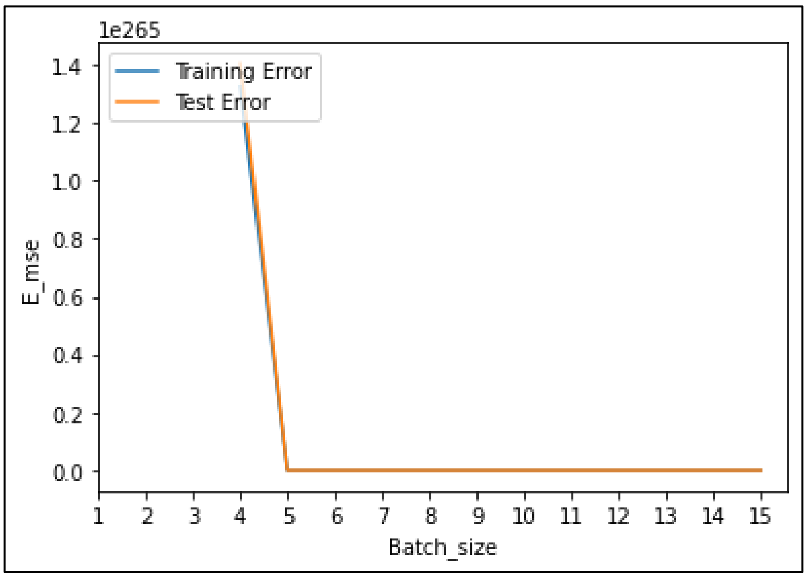


Fig 6. The error for batch size < 4 was out of bounds. Nonetheless, it converges

Therefore, our initial batch size value of 10 was good for our final estimate of the polynomial which is:

$$\boxed{8.88472159 + 0.8518583\, \boldsymbol{X} - 1.09412644\, \boldsymbol{X}^2 + 0.62121061\, \boldsymbol{X}^3 + 2.52004379\, \boldsymbol{X}^4 - 0.22660583\, \boldsymbol{X}^5 + 0.22836758\, \boldsymbol{X}^6 - 2.82967896\, \boldsymbol{X}^7}$$

- <u>Using all 100 data points</u>:

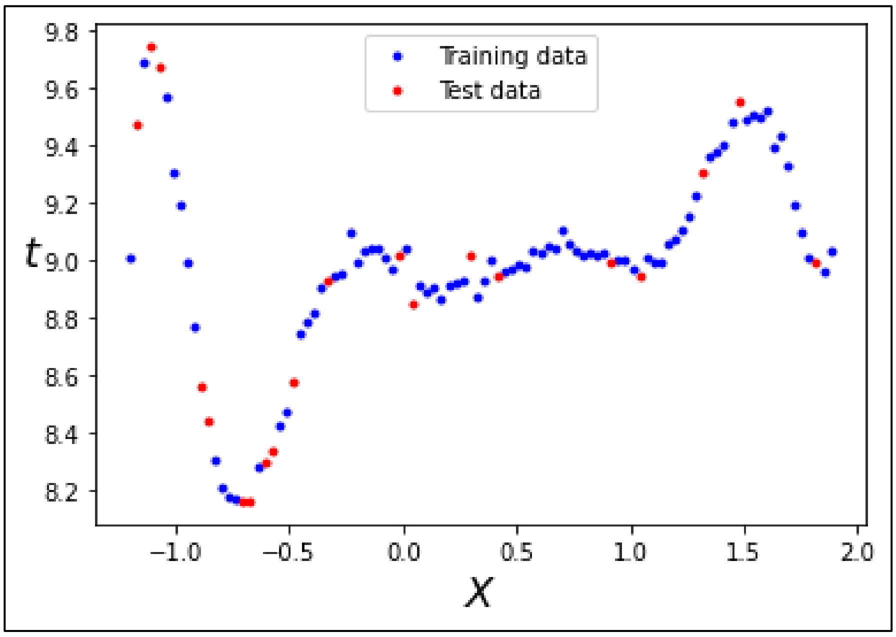


Fig. 7

- **Using Moore-Penrose pseudoinverse:**

I run my code for different values of λ (regularization constant) and epochs (number of iterations). [Fig 8.]

Once again, we observed that the number of epochs have no effect on the losses. Thus, I iterate over different λ. We can also see that while the training error has an elbow point at m=5 (m is the highest degree of polynomial), the test set is truly reduced for m=9. Thus, for further iterations, I will fix m at 9. [Fig 9]

This time, we see that since number of data points is 100, there is some improvement with regularization at λ = 0.01 after which it starts under-fitting.

A final plot for λ = 0.01, number of epochs = 1, over different values of m is shown in Fig. 10 which confirms m= 9 to be a good basis.

Therefore, the best guess of the polynomial, for m = 9 and λ = 0.01 is:

$$9.0140234 + 0.01824327\,\boldsymbol{X} - 0.97251592\,\boldsymbol{X}^2 + 2.73209287\,\boldsymbol{X}^3 - 1.05359651\,\boldsymbol{X}^4 - 4.21682231\,\boldsymbol{X}^5 + 4.04789536\,\boldsymbol{X}^6 + 0.86240564\,\boldsymbol{X}^7 - 1.9421145\,\boldsymbol{X}^8 + 0.51181432\,\boldsymbol{X}^9$$

Finally, Fig 11 shows this polynomial on data-points from the dataset

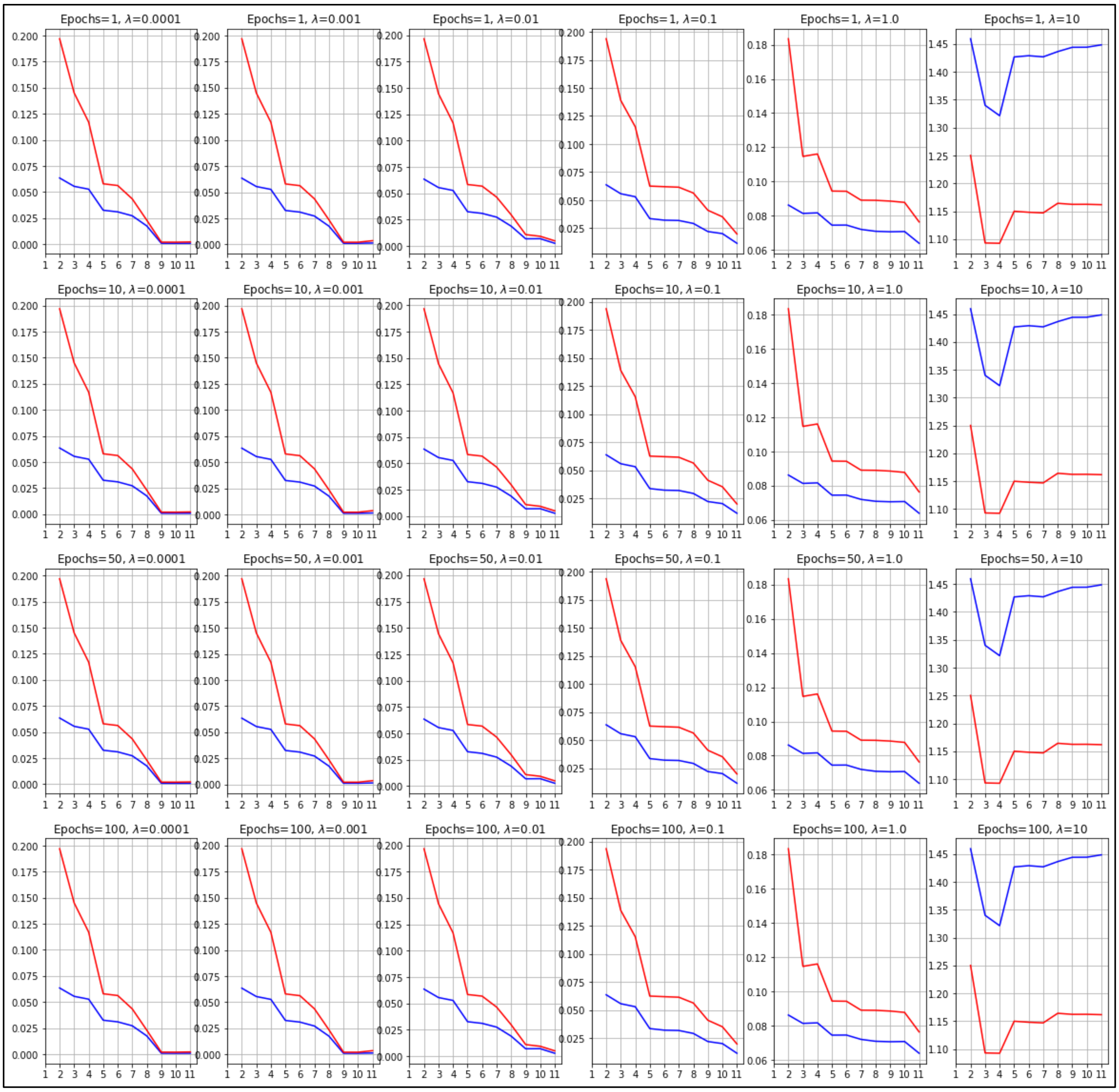


Fig 8. The red line depicts the test error and the blue line shows the training error

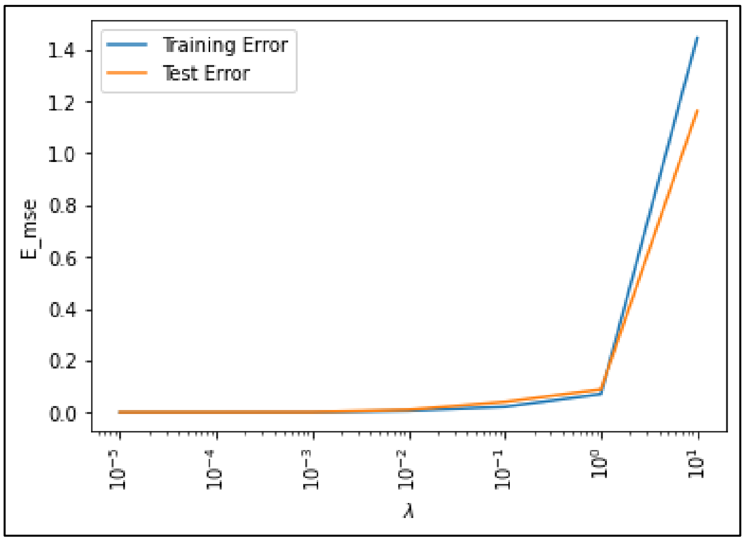


Fig. 9

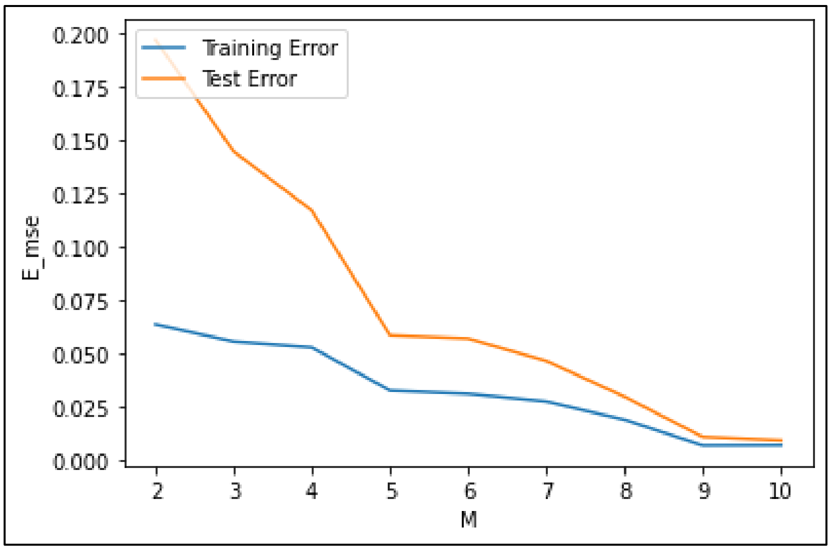


Fig 10.

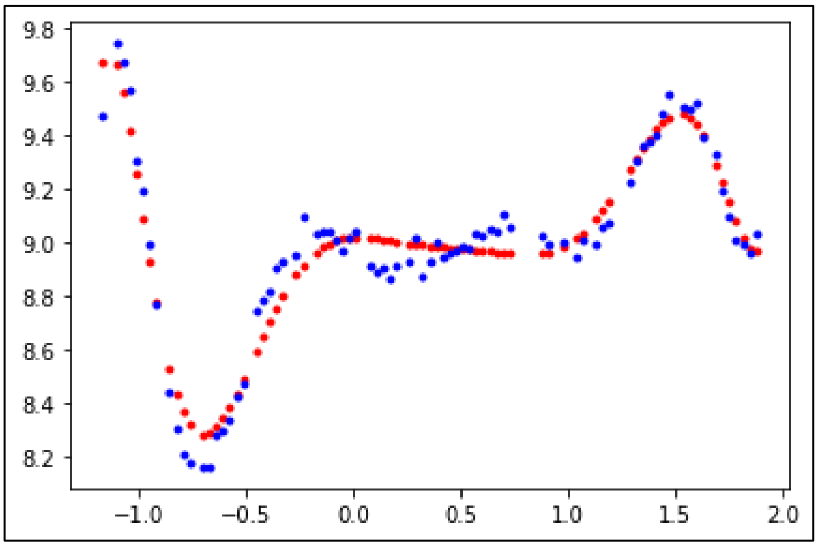


Fig 11. Red dots showed the predicted target labels and blue dots are from the dataset

• Using Gradient Descent:

I followed the exact same procedure as listed for the 20 data points. However, I will need to check for appropriate value of λ as well.

First, for the learning rate, after trying various different values for η, I came to the conclusion that there is no single good value for the learning rate and that it needs to change with time. Thus, once again after extensive experimentation, I found the value of $\frac{0.3}{i+1.7}$ to work best for the given data. Here, "i" is the starting index for a certain mini-batch.

For the order of polynomial, I take number of epochs and lambda, and do mini-batch gradient descent on it for a batch size of 25. The results are in Fig. 12.

From Fig 12, we infer that the gradient descent seems to approximate the model to its best at M=5. Also, the model seems to perform well for $\lambda = 0.0001$ and epochs = 2000

Note: For all models of gradient descent, I have used a min-max normalization followed by subtracting each dimension by its mean for faster and better convergence (Centred around 0). The same normalizing parameters (Minimum, maximum, mean of min-max normalized training set) is used to normalize the test set

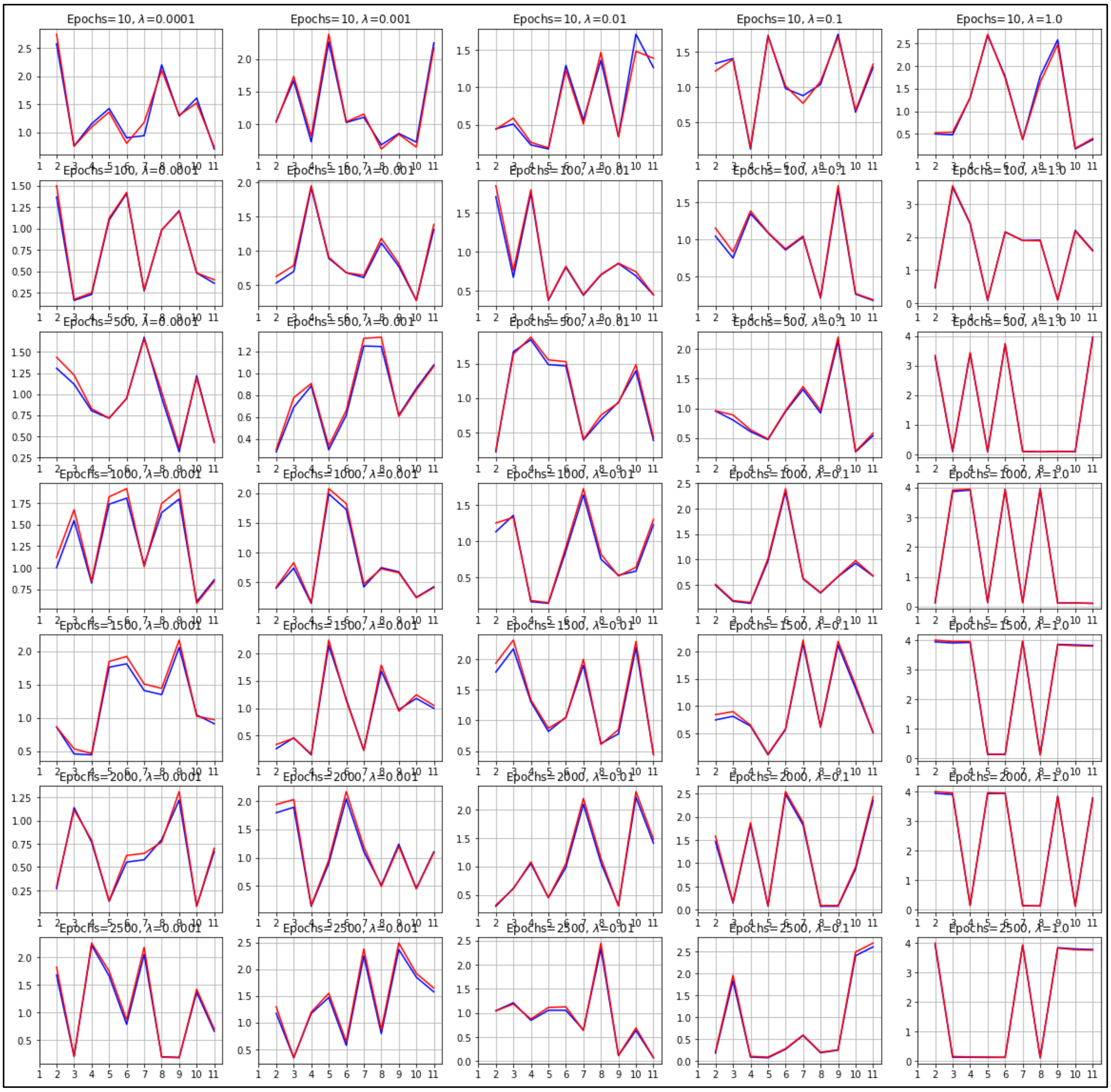


Fig 12. Blue is training error and red is test error

Next, I plot the errors for different values of batch size from 1 to 80 (Size of training set) [Fig 13.]

Both batch gradient descent and SGD seem to show good performance (Batch being better), but for few values in the middle, the accuracy decreases. Since there is sharp decline around 32, we take that to be our batch size for final evaluation of degree of polynomial [Fig 14.]

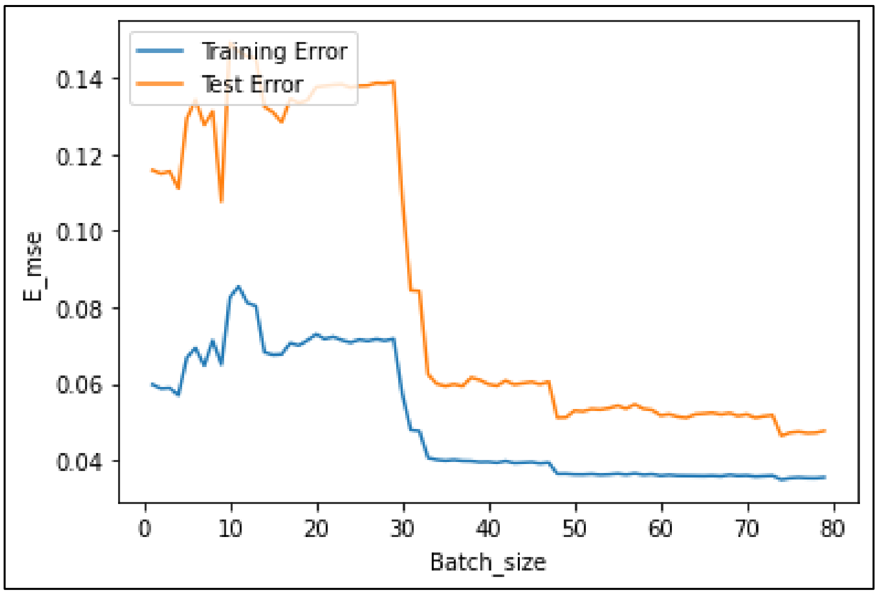


Fig 13.

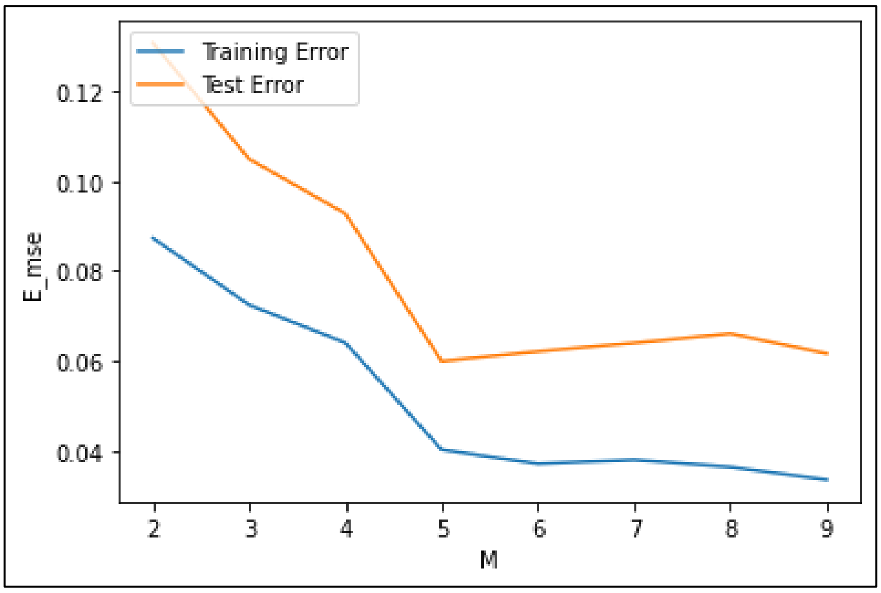
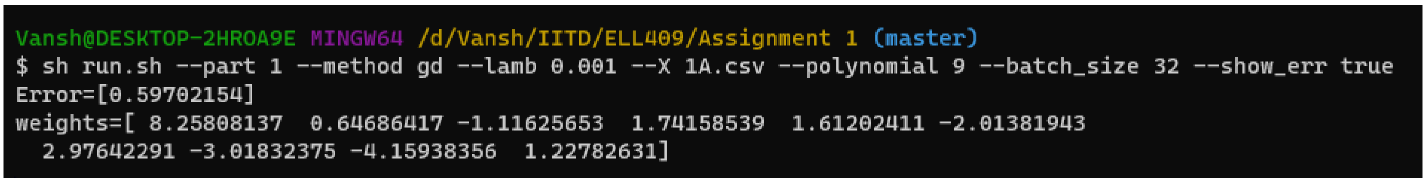


Fig 14.

Therefore, our initial M value of 5 was good for our final estimate of the polynomial which is (For λ=0.0001, epochs = 2000, batch size = 32):

$$\boxed{8.97359663 + 0.59243457\,\boldsymbol{X} - 0.0.23539727\,\boldsymbol{X}^2 + 2.35287087\,\boldsymbol{X}^3 + 0.51712917\,\boldsymbol{X}^4 - 3.44087613\,\boldsymbol{X}^5}$$

A value of M=9 (Same as pinv) for batch size = 32 naturally works better since we report the training MSE and that should decrease with the highest degree. The corresponding weights are:


```
Vansh@DESKTOP-2HROA9E MINGW64 /d/Vansh/IITD/ELL409/Assignment 1 (master)
$ sh run.sh --part 1 --method gd --lamb 0.001 --X 1A.csv --polynomial 9 --batch_size 32 --show_err true
Error=[0.59702154]
weights=[ 8.25808137  0.64686417 -1.11625653  1.74158539  1.61202411 -2.01381943
  2.97642291 -3.01832375 -4.15938356  1.22782631]
```

Anyway, my SGD shows no significant improvement from M=5 to M=9 and thus can give similar approximation for the curve. The corresponding plot for M=5 is:

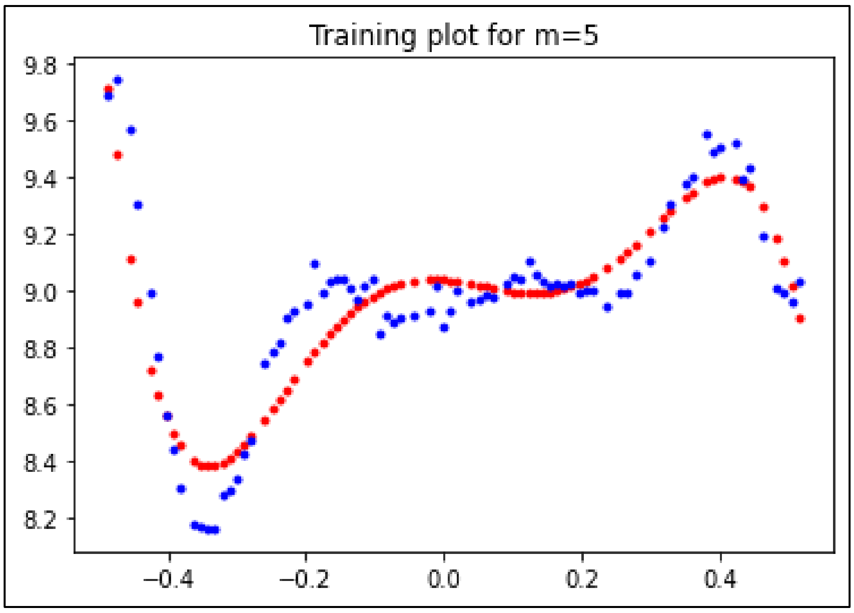
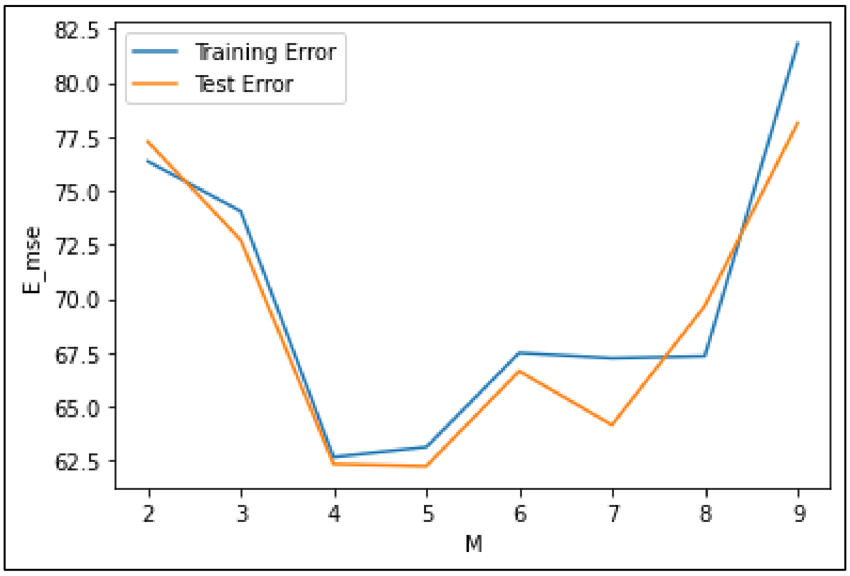


Fig 15. Red dots are predicted values, blue dots are from dataset

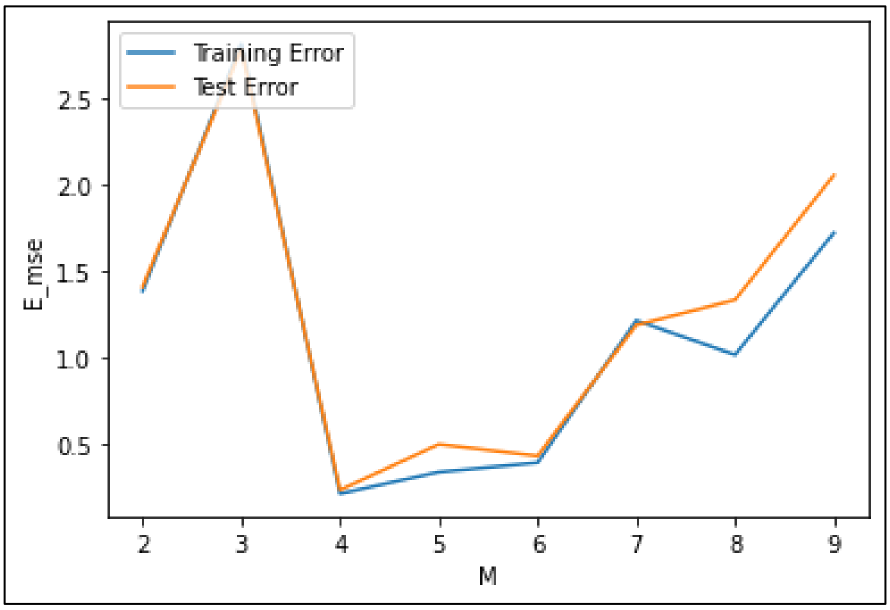
Increasing the number of iterations/epochs did not increase the accuracy by much, and the results were pretty random (as seen in Fig 12). In normal sense, we would expect something of a convergence with number of epochs, at least upon the training set. One possible reasoning for this is that my learning rate depends upon batch size index and not the iterations, so overfitting on the data is not as obvious. However, changing that learning rate decreases the model's performance. Another reason can be that the model was dwindling between crossing the minima and coming back to it as it saw the data again and again.

However, I was still not satisfied with the performance. So, I tried a couple of other things to increase the accuracy of my model. The plots for all of those are not provided in this report for all the experiments, but they are included inside the code that I submitted along with this report.

The first thing that I did was to prevent the problem of exploding gradients. I tried to clip my gradient and treated the max and min values as hyperparameters ranging from 0.5 to 1 (And consequently -1 to -0.5 on the negative side). This however, increased my running time by a lot (Had to interrupt the operation after 20 minutes). This could have been because the gradient and learning rate were both being restricted as time passed so I tried some other combinations of these. Constant learning rate with clipped gradient, and unclipped gradient with dynamic learning rate (My original experiment). Here is the plot for clipped gradient and constant:



Next, I normalized my gradients before returning them. Once again, to mitigate the issue of exploding gradient. This gave me a good performance and my weights stopped blowing up to nan on slight perturbations in the externally provided parameters. I used an L2 normalization



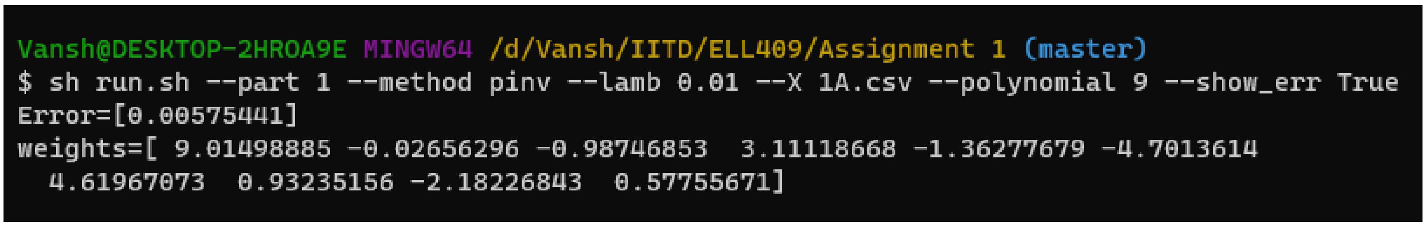
Even though the performance is poorer than my normal method, it prevents the weights from going to infinity by just a line's addition to the code. I get slightly better performance by tweaking the dynamic learning rate.

Conclusion

Thus, we conclude that increasing the number of data points gives a much better model prediction. However, for a gradient descent model to work, we would still require much more data points.

In the case of pseudoinverse, models for both n=20 and n=100 were a 9th degree polynomial with closely resembling plots. But for the stochastic gradient descent, the degree was different and the error was relatively more. Since the degree was different in this case for n=20 and n=100, naturally we saw a change in number of iterations (epochs) required. Another thing to note is that since the target values are really high for part B, so is its error even for close approximations. It is still reduced by a lot if we decrease our regularization parameter, as it allows my model to fit better using gradient descent. We also see that regularization doesn't help much with 20 data points probably because of the presence of noise and the model's inability to fit on the data. For 100 points on the other hand, regularization prevents overfitting at even higher degrees.

Finally, the estimate of the model (By the Moore-Penrose pseudoinverse method on all 100 data points with polynomial 9 and λ=0.01) can be taken as:



```
Vansh@DESKTOP-2HROA9E MINGW64 /d/Vansh/IITD/ELL409/Assignment 1 (master)
$ sh run.sh --part 1 --method pinv --lamb 0.01 --X 1A.csv --polynomial 9 --show_err True
Error=[0.00575441]
weights=[ 9.01498885 -0.02656296 -0.98746853  3.11118668 -1.36277679 -4.7013614
  4.61967073  0.93235156 -2.18226843  0.57755671]
```

- ## Noise Estimate:

  I have not calculated noise estimate for each of the 4 models, but only for Moore-Penrose pseudoinverse for all 100 data points

  How to estimate variance of the noise?

$$y_i = f(X_i) + \varepsilon_i$$

$$y_i \mid X_i \sim \aleph\,(f(X_i), \sigma^2)$$

$$Var(Y) = E\,[(Y - E(Y))^2]$$

$$\text{Where } E(Y) \text{ is our estimate of } f(X)$$

$$= 0.00694036 \text{ (By the model trained on just training set)}$$
$$= 0.00575441 \text{ (By training on entire dataset)}$$

  Therefore, variance of the underlying noise is somewhere around **0.00575441 – 0.00694036**

  It should be noted that as the number of data points increase, the variance in noise should also increase as more data points mean more noise to get captured but at the same time, our model will be better at predicting these values, thus decreasing our accuracy to be able to calculate the variance exactly

Part 1B

For this part, my focus will be on noise and plots and highest degree of polynomial, as suggested in the assignment pdf. Therefore, I will only be using the Moore-Penrose Pseudoinverse for determination of the underlying polynomial. First, we look at the plot

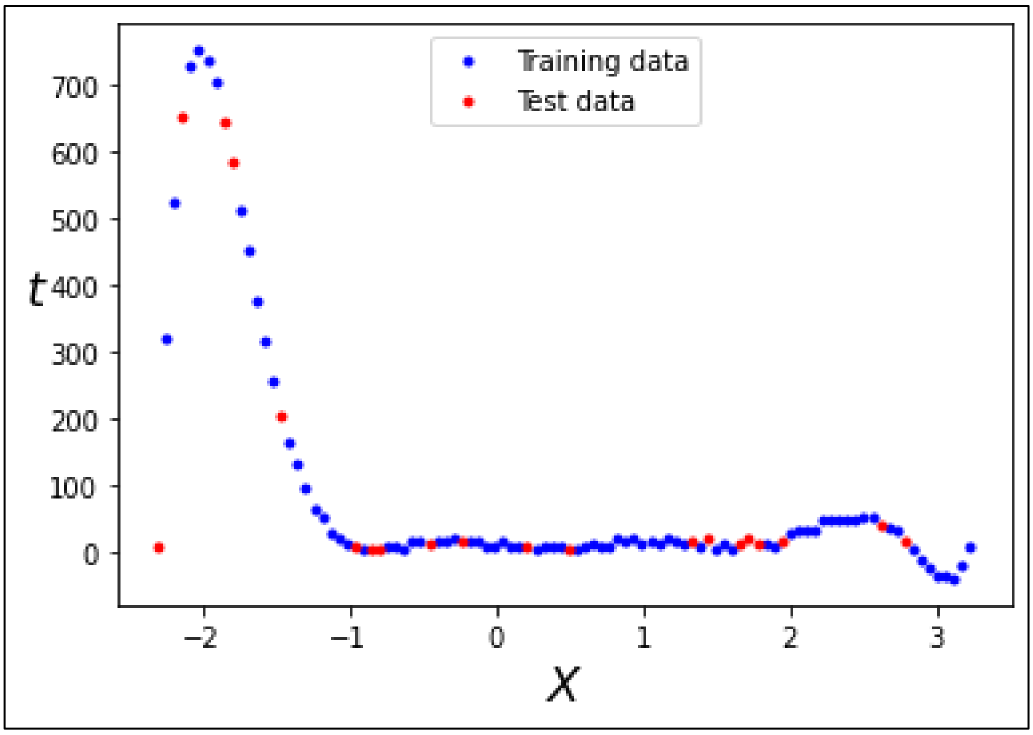
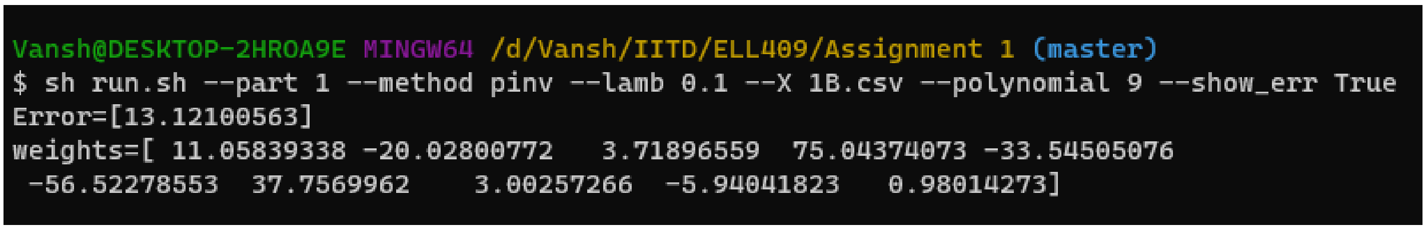


Fig 16.

Now, we run it through different values of the regularization constant for different values of the degree of polynomial to find further inference. [Fig 17]

Clearly, we see that M=9 is great, but we still need a better plot for λ. [Fig 18.]

From this we see that for $\lambda > 10^{-1}$, our model starts to underfit the data. The final polynomial using M=9 and $\lambda = 10^{-1}$ is:



```
Vansh@DESKTOP-2HROA9E MINGW64 /d/Vansh/IITD/ELL409/Assignment 1 (master)
$ sh run.sh --part 1 --method pinv --lamb 0.1 --X 1B.csv --polynomial 9 --show_err True
Error=[13.12100563]
weights=[ 11.05839338 -20.02800772   3.71896559  75.04374073 -33.54505076
 -56.52278553  37.7569962    3.00257266  -5.94041823   0.98014273]
```

The plot for the same can be seen in Fig 19.

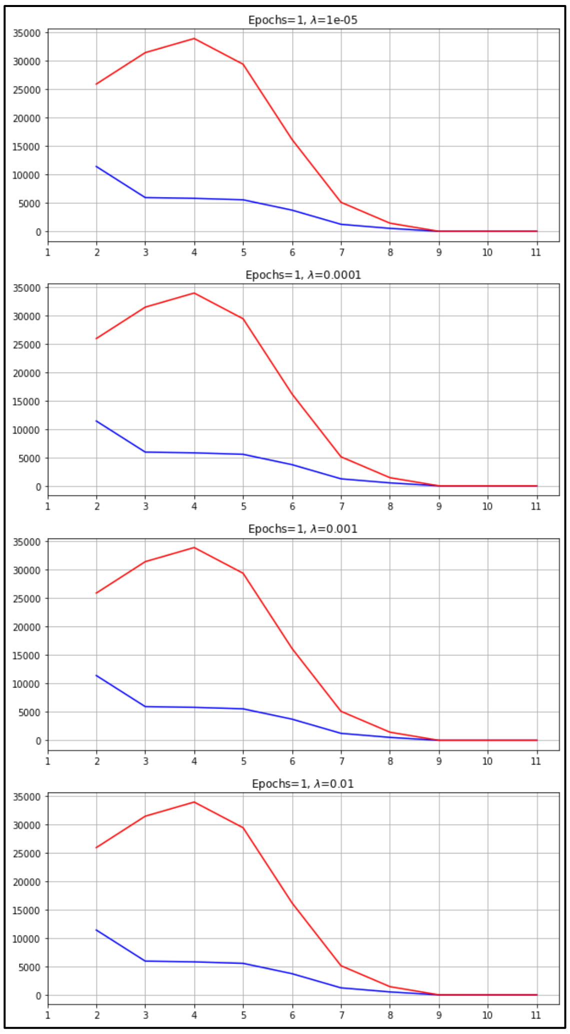


Fig 17.a

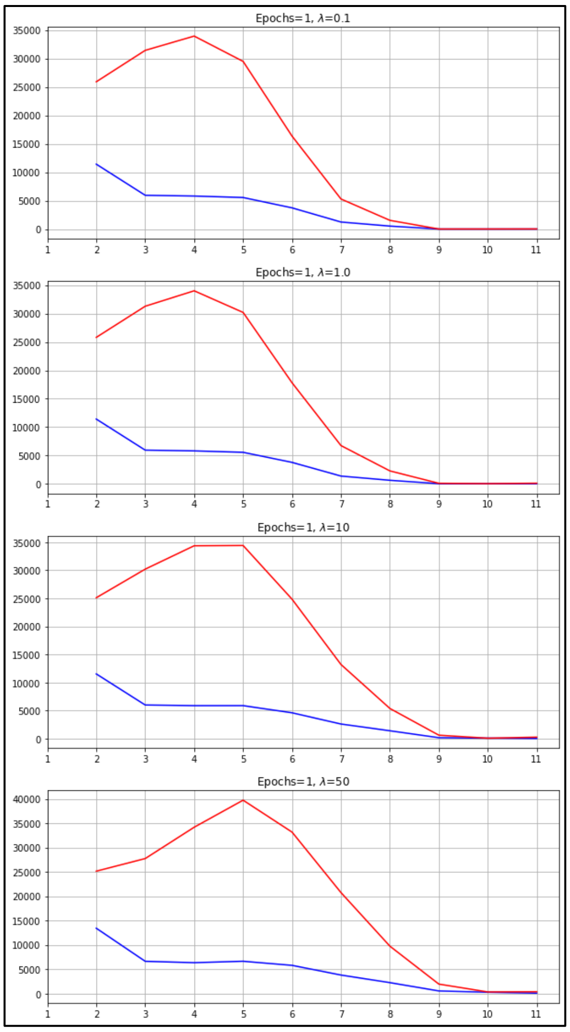


Fig 17.b

y axis depicts the MSE error and x axis shows the different values of m. Blue line is for training dataset while the red line is for testing dataset

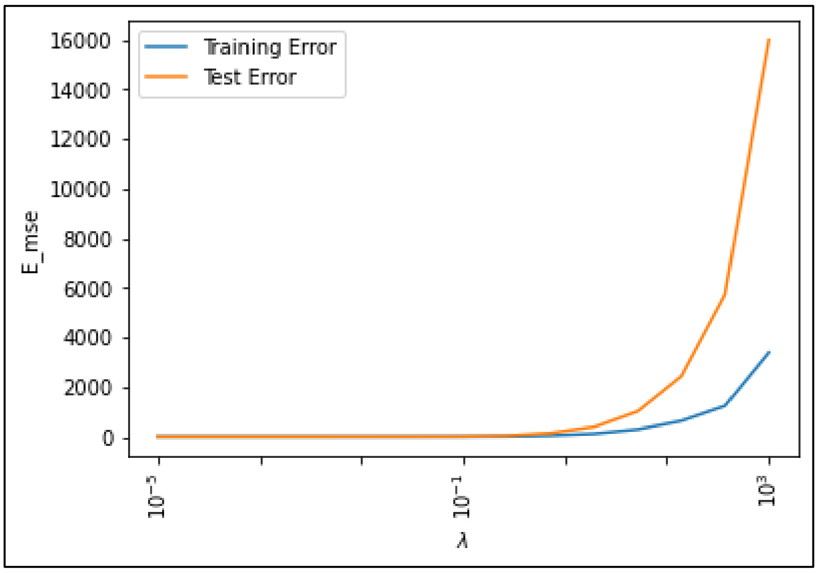


Fig 18.

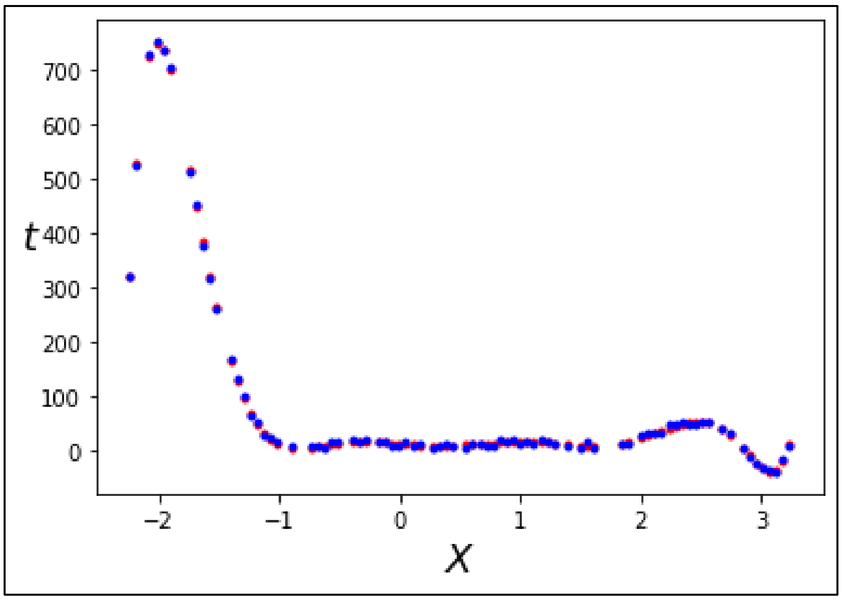
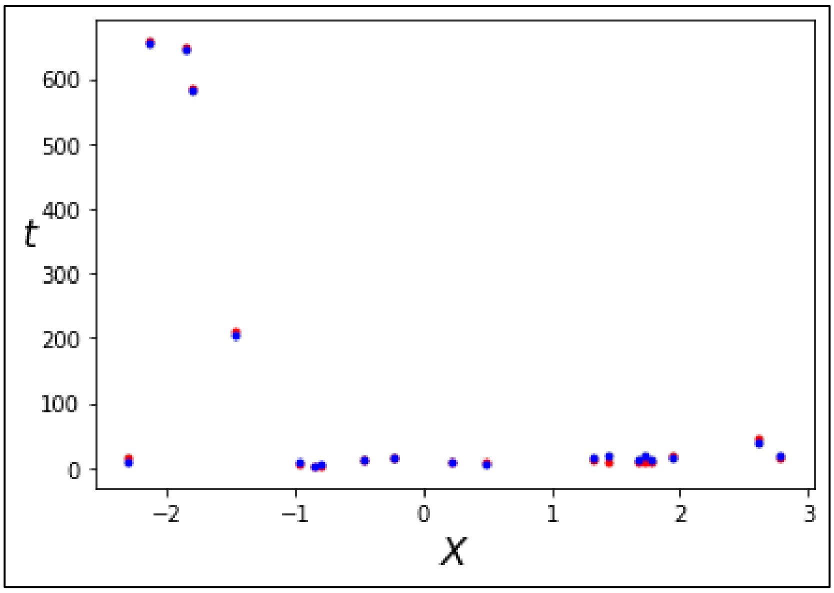


Fig. 19. Red dots are predicted positions and the blue dots are from the given data

The corresponding plot for test set is:



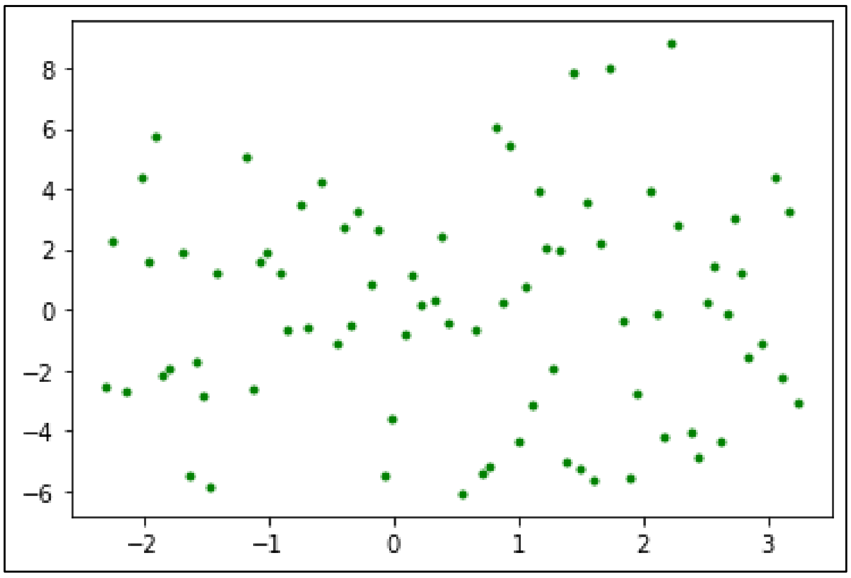
Since the predictions are not far from the point, we can say that the model is not overfitting.

Noise Estimate:

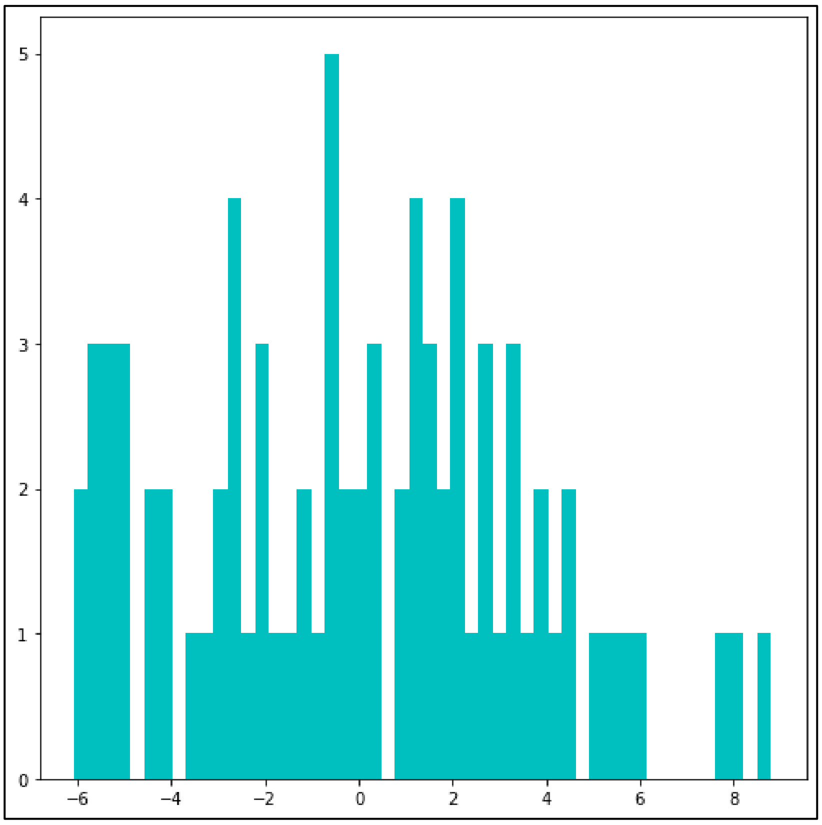
Let’s assume a dataset (x, f(x) + ϵ) where f(x) = g(x) – c and ϵ = c
Here, ϵ was the noise in my dataset (equal to a constant value c). Thus, finally do we consider it a dataset as shown or simply (x, g(x)) which is what it sums up to. i.e., a dataset without any noise.

What I intend to convey is that where no information is given about the noise, I cannot truly discover what kind of probability distribution function this noise is following.

Anyway, I found out f(X) – y where f() is the function that I fit on my data. I plot these points as a function of X to see if I can infer anything:

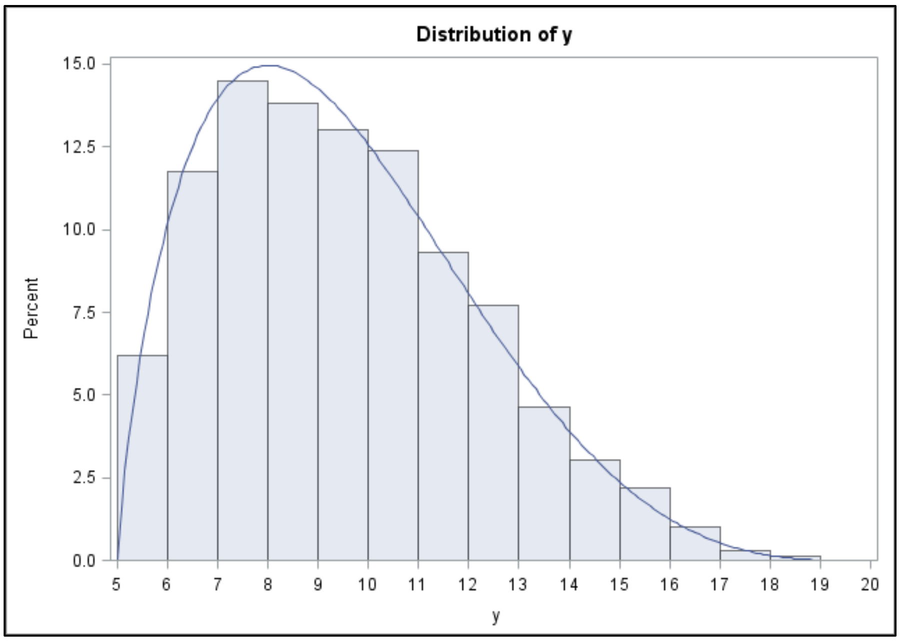


This seems quite random. But then I plotted its histogram with 50 bins:



Checking out various distributions and their curves from https://www.itl.nist.gov/div898/handbook/eda/section3/eda366.htm

I believe that the noise is from a **beta distribution**. For reference, here is a plot of **beta distribution** along with its histogram:



I chose a beta distribution over a gamma distribution because of the skewness (tail in distribution) in the beta distribution with tail towards the right side, which is what I noticed for my model too

Miscellaneous:

One thing that I kept constant throughout all my experiments with gradient descent was the cost function, the derivative of which drove my weights in the right direction. So I decided to tweak this too. I first took a power of 3 instead of 2 from MSE. The results were quite erratic which is unsurprising as the signs are also included in this form of weight updation. Then, I tried with a value of 4, and there wasn't much change. However, once I removed my weight normalization step, I realized that the exploding gradients problem came sooner with such an error function and it was quite out of control as finding the minima became almost impossible in this setting.
Finally, I tried the MAE method (Again commented out in the code) which did show decently similar performance but that was a little risky to use as it is not differentiable around 0. Moreover, one big problem in using MAE loss is that its gradient is the same throughout, which means the gradient will be large even for small loss values. This isn't good for learning.

I therefore conclude part 1 of the assignment with all my experiments for the reader to go through.

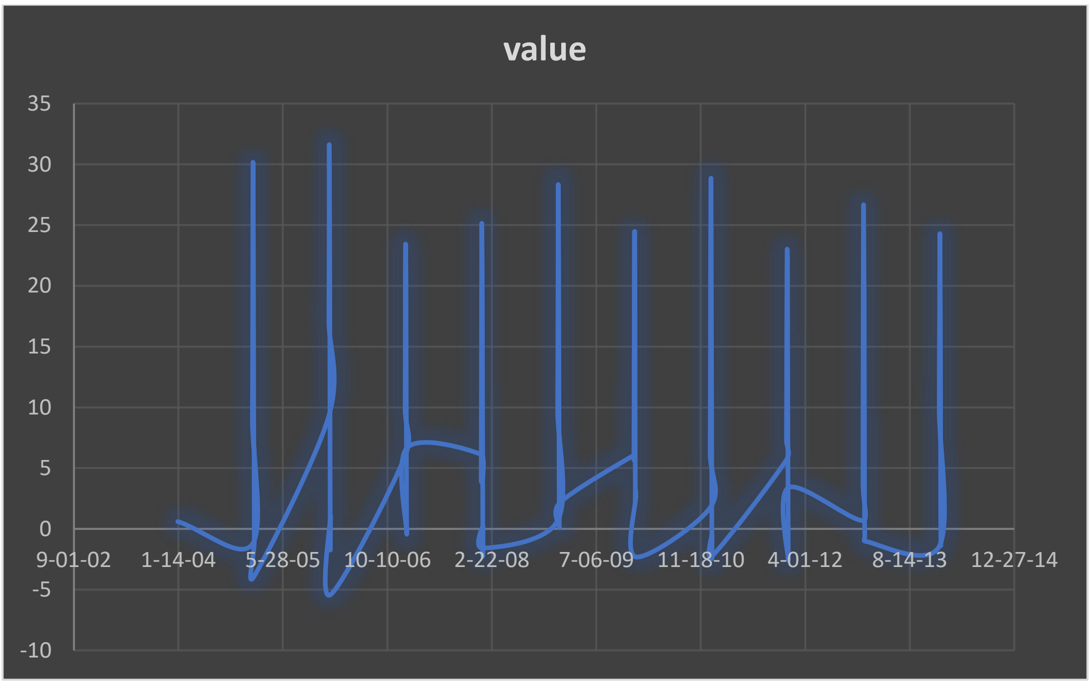
Thank you
Vansh Gupta

Part 2

My first submission to the leaderboard, was well before looking at the data. I simply took the data, added some degrees to all of it and put it up checking the least error. It gave me a decent score, but it was not something fancy that I should discuss here (Even if I am not able to reproduce the results)

The first thing that I did, before having a look at the data, was convert the discrete points of data and month as one single continuous variable by min-max normalizing the date and month and adding the two values (Let's call this new value dime). After that, I tried a bunch of different models with dime and year, and their various exponentials, all of which gave me error to the north of 30. Clearly, I was overcomplicating matters as earlier, my basic submission fetched me good score. (All this is backed by the code that I submitted. This is why I have not provided plots for these, as they can be checked in the python script and is pretty much useless)

Then, I plot the given data on excel, to find this:



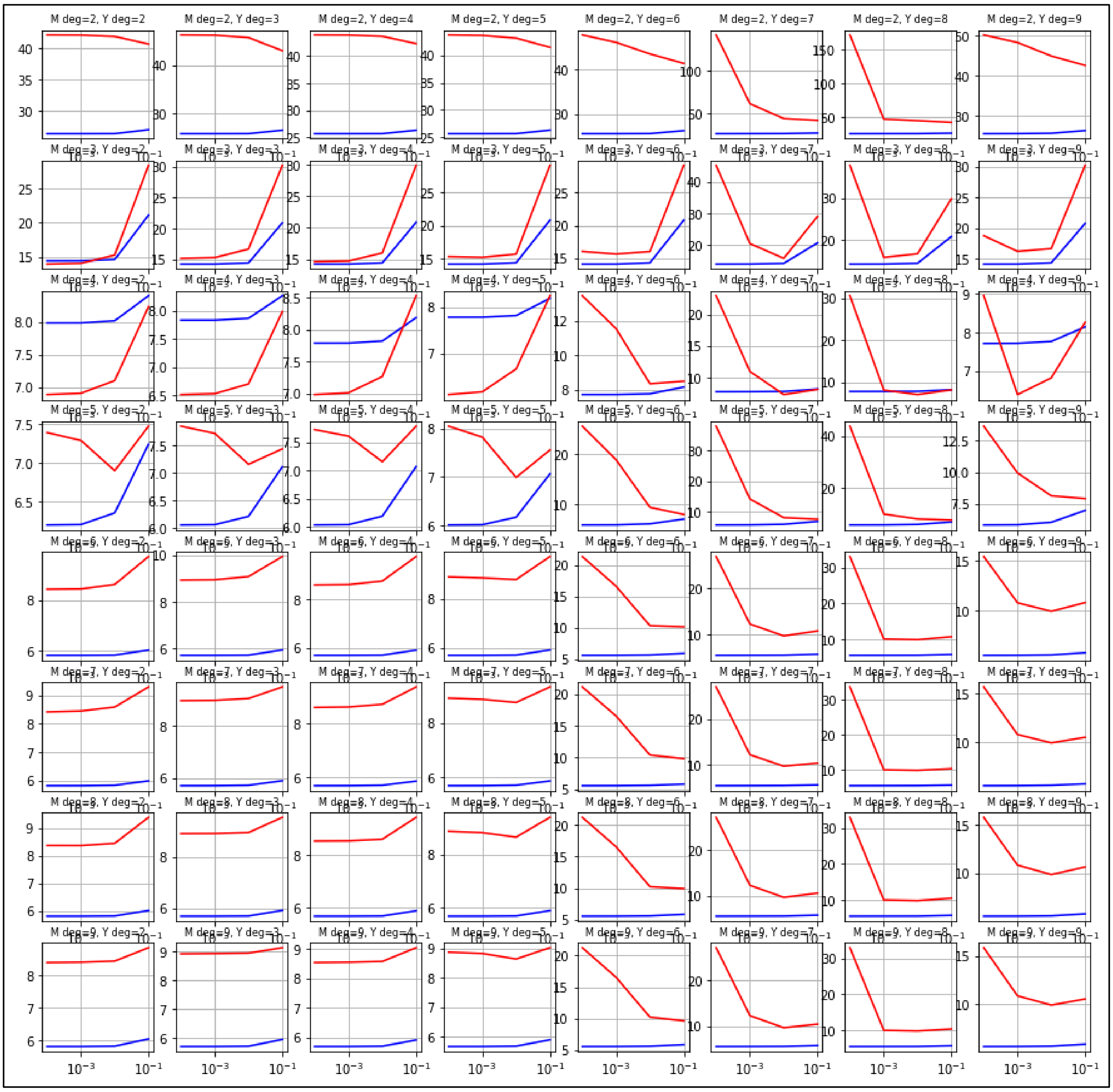
There are 3 things that one can infer from this procedure.
1. My excel skills need some work
2. All the dates were 1st date of a month of some year, and will thus have no bearing on the final values. i.e., these can be discarded, thus reducing the given parameters to month and year, and simplifying our model
3. Lastly, we see that the model is repeating similar trends in a periodic manner and upon further investigation, I found that the model was periodic in terms of year. Hence, multiplying month and year would not have helped with the predictions.

At this point, I was guessing it was data of avg sales per million or some other value for something sold more in summers, but given the negative values, it could simply be avg temperature on a particular day, which would explain the trend

Nonetheless, I knew what I had to do. Beautify the data by splitting id to month and year, treat them as separate values because their multiplication would make no sense as they seem pretty independent and also, we can apply our prior here that years and months are not correlated.

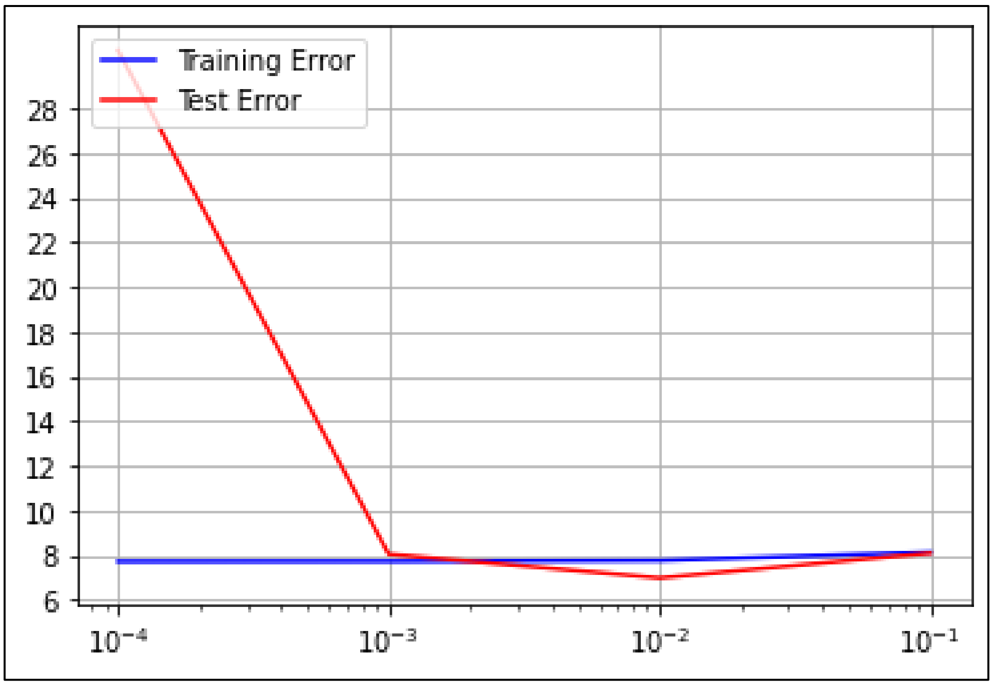
Finally, due to the lack of bigger dataset, I decided to go with the Moore-Penrose pseudoinverse method of finding my parameters, because my previous analysis showed that it gave better solutions in lesser time for fewer data points (and also because we were only allowed to use linear regression here)

Note: Test set for most part refers to the validation set

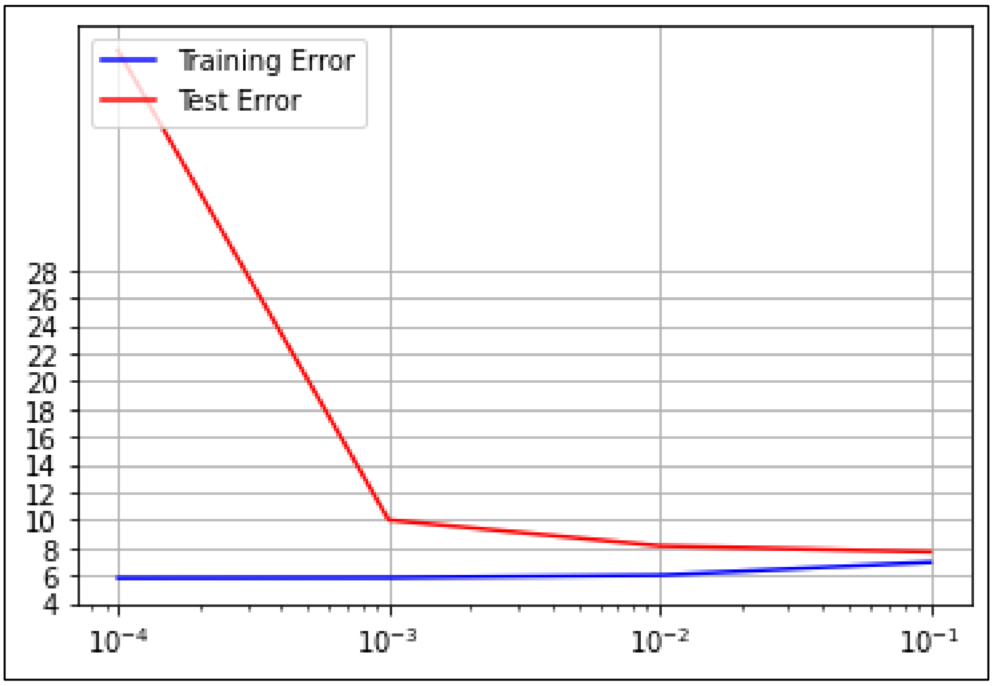


The degree of both month and year columns ranges from 2 to 9, and λ is from [0.0001, 0.001, 0.01, 0.1]

We can see that the highest degree of year 8 gives good performance for highest degree of month as 4 and 5. Plotting these 2 across different λs gives:



Month degree= 4



Month degree = 5

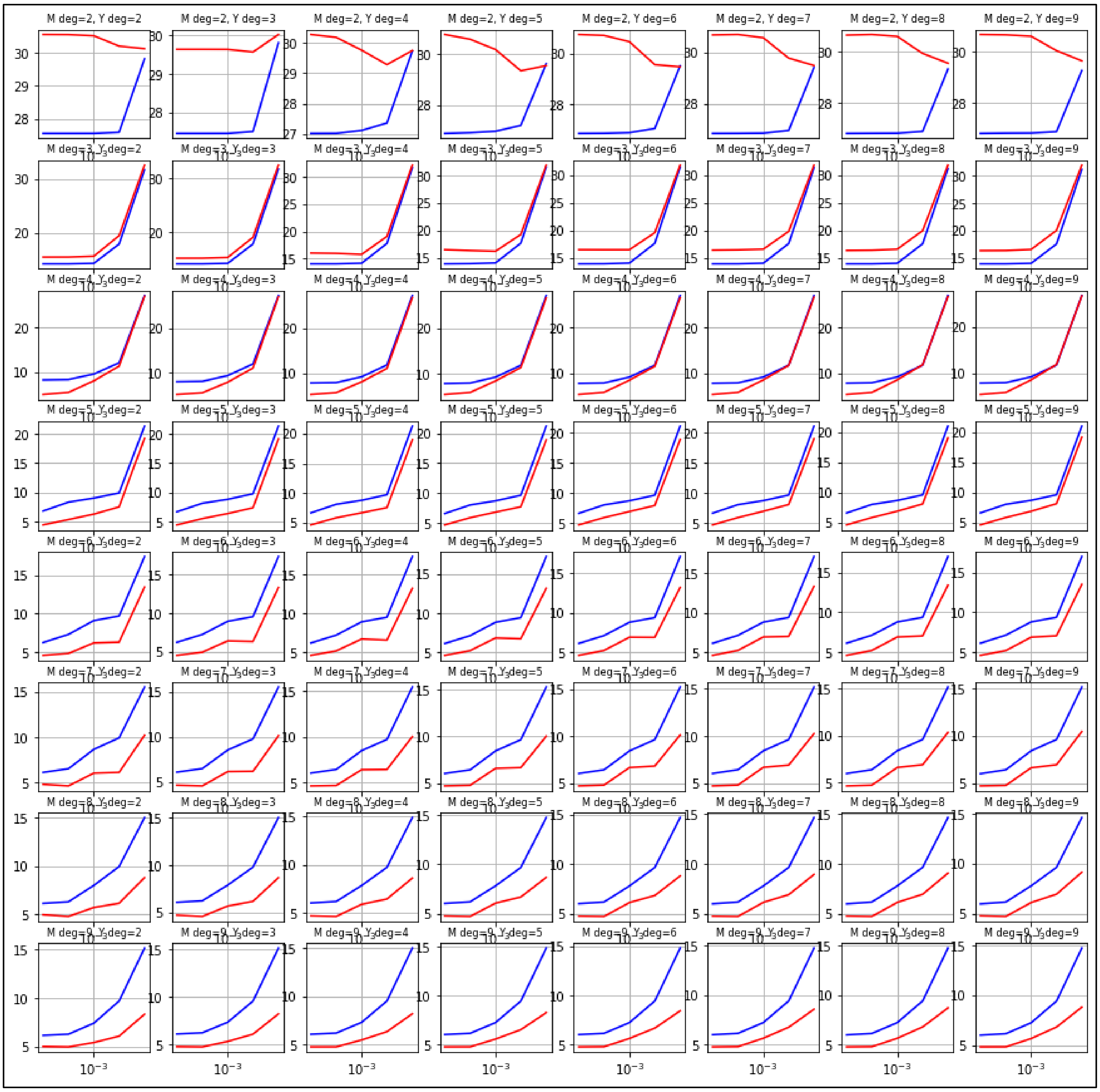
Value of λ can be 10^{-2} with month degree = 4. Corresponding values of MSE are:

Training set = 7.765861574815575
Dev set = 6.988325949280505

At one point, while squaring and cubing and … the columns, I was raising them to some power of 10 to bring to meaningful form (Not become insignificantly small). Upon seeing the final arrays, I realized some of them were blowing up to very large values. So, I repeated my experiments without that to get optimal value at $\lambda = 10^{-4}$ and (Y, M) = (3, 7). Corresponding values of MSE are:

Training set: 6.232642519793618
Dev set: 4.5941934085544665

Corresponding plot for different M and Y:



This submission improved the score by a lot and brought me to the range of 6. I was still pretty behind on the leaderboard, but now nothing more could have helped much. I still tried finetuning λ a bit more but the values I received were far from the values that got me the score of 6.

Note: The values I was submitting were trained only on 90% of the data, and not the entire dataset. My final submission will be trained on the entire dataset

Finally, after testing my model in a k-fold cross validation setting, my final values were still pretty close to the submission that gave me a 6.xx score. On looking at the leaderboard I observed that the top 4-5 positions had the same score for an MSE based evaluation. And since everyone would be facing similar problems as me with accuracy in gradient descent, they must have used pseudo-inverse for their predictions.

There can be 2 reasons for this discrepancy in our scores (After convincing myself that a product of month and year will not improve the performance):
1. I never considered a power of 1 i.e., the model could be even simpler
2. They were using something which wasn't exactly machine learning

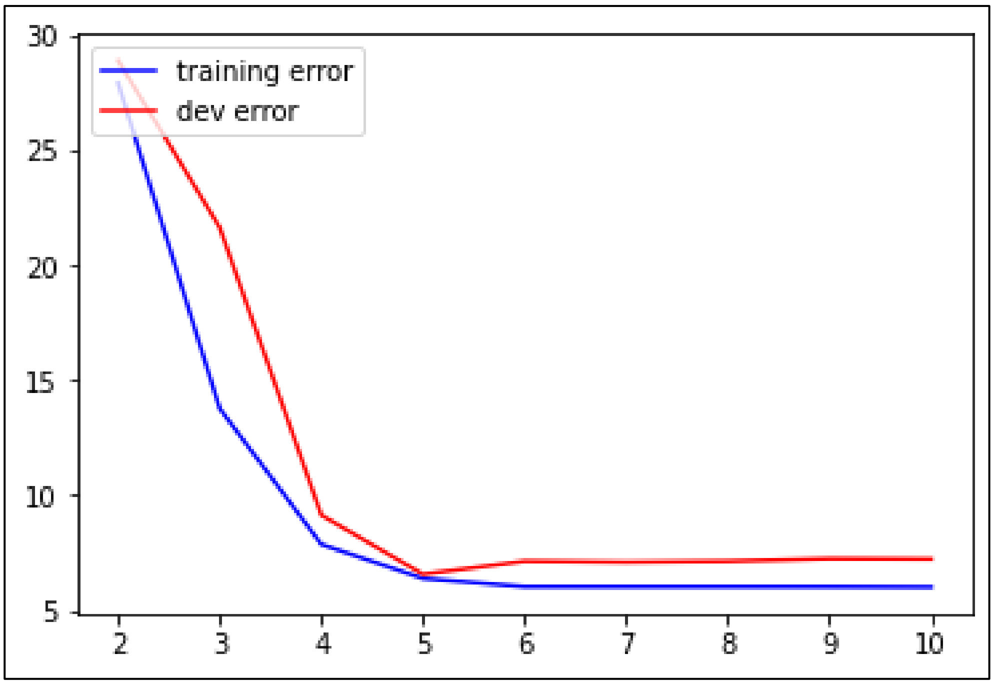
Another observation at this point was that since the data was repeating in a yearly fashion, I need not consider even "year" as a variable. Leaving me with just month as my data point, a bias, and **maybe** single power of year for gradual trend

I decided to run two more models, one was just average of the previous instances for that month (In a formal setting, I can call it a prior or more concretely, a moving average model but the latter is used for stationary data where we have previous instances)
The 2nd model is the one as I described above.

A very small piece of code gave me really good training and dev errors.

The 2nd model also gave values very close to the ones by the previous model for different powers of month ranging from 7 to 12 so I took the one for M=7. My score on test set improved.

One more thing I did was to remove that "maybe" from few lines back and only treat my month as a parameter. I plotted training and dev set error for different values of highest exponent to get:



Therefore, I set my highest exponent to 5 to generate a very simplified model compared to my initial iterations

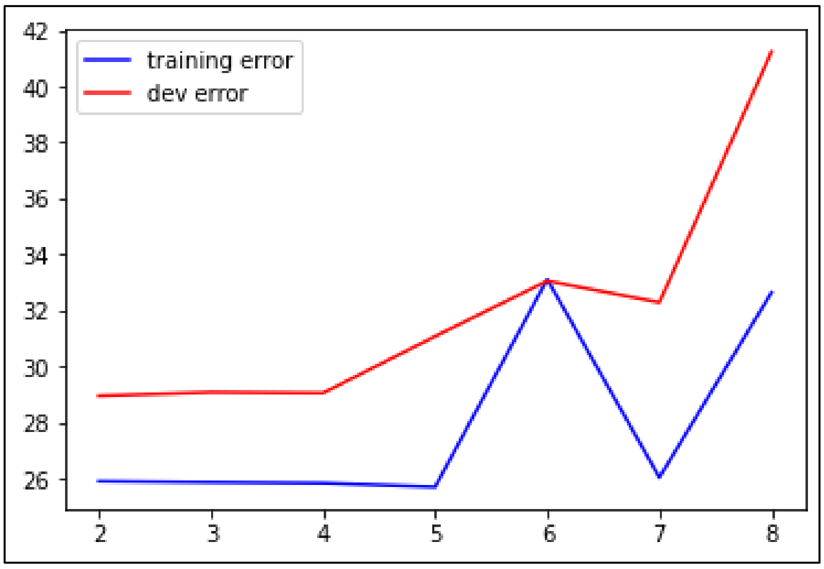
However, I submitted both of these sets of values and was able to reproduce the same values as the model which was simply average of all instances of that month by setting M=12. This gave me a score of 5.15135 and the other model gave me a score of 5.4xx and another with differently normalized that gave me a score of 5.13xx

Since my score was now also the same (5.15135) which means, I was right in predicting their method. If I was an ML model, my parents would have received best paper award for creating me. Also, I am being too informal in this report. My final submission, which was also my best score was simply average of my two best models

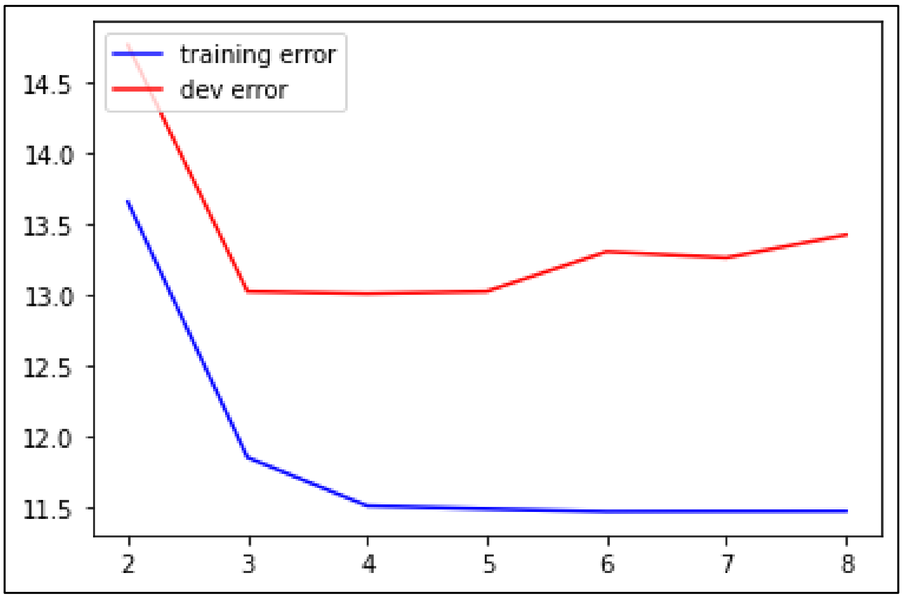
I had one last game left in me. Therefore, I visited the graph again, realized that the values were high for months in the middle, and therefore chose my basis function as:

$$\sin((month - 1) * (\pi/11))$$

The idea was to bring a non-linearity in the model by adding it manually instead of expecting the model to learn this. I used a k-fold cross validation for this but the results were not quite what I expected (Y axis denotes the MSE and the x axis shows the maximum power of this variable. Same for the plot that follows)



Used a similar idea for the basis function $|(month - 6)|$, but again, the results were disappointing:


training error
dev error