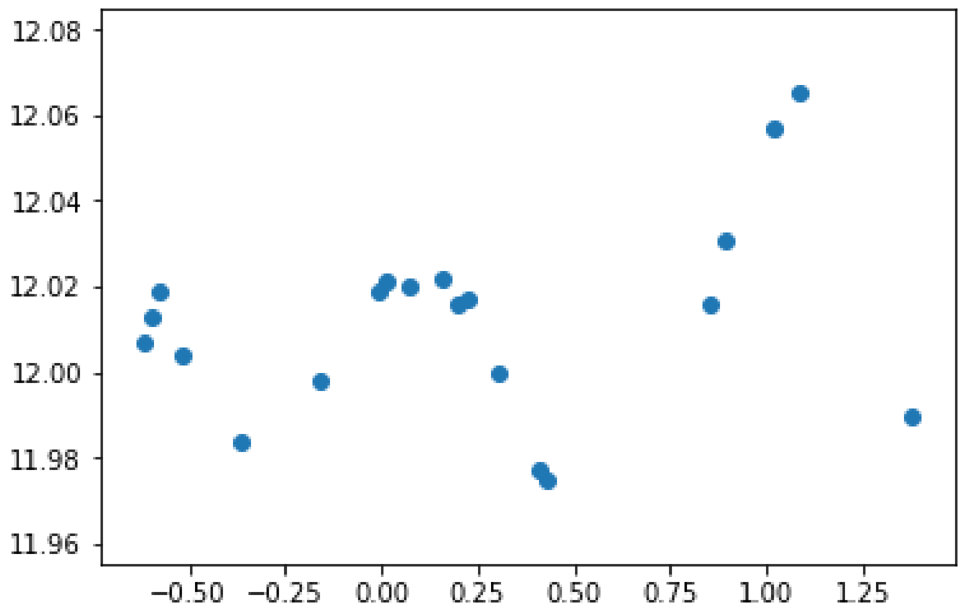
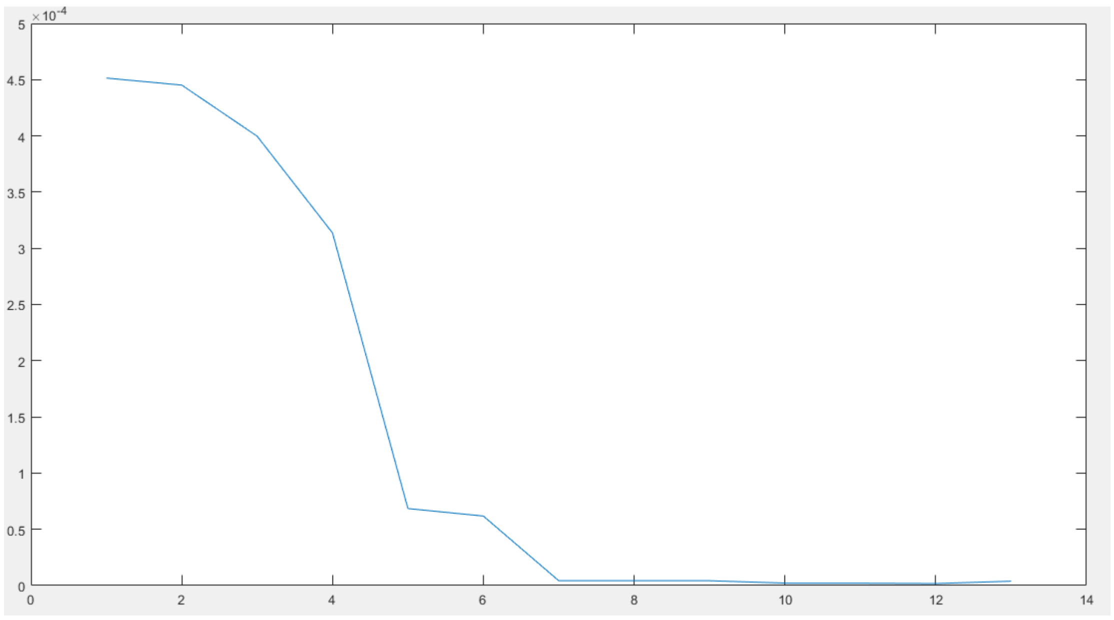
PART 1

Visualising the data(first 20 points)



To start off, we first try to get an estimate of m, for which we use the Moore-Penrose pseudoinverse function. We vary m from 1 to 20, create the design matrix, get the weights and the corresponding error, which is plotted below as a function of m. In this, we have used validation and split the data in a 80:20 train to test ratio.

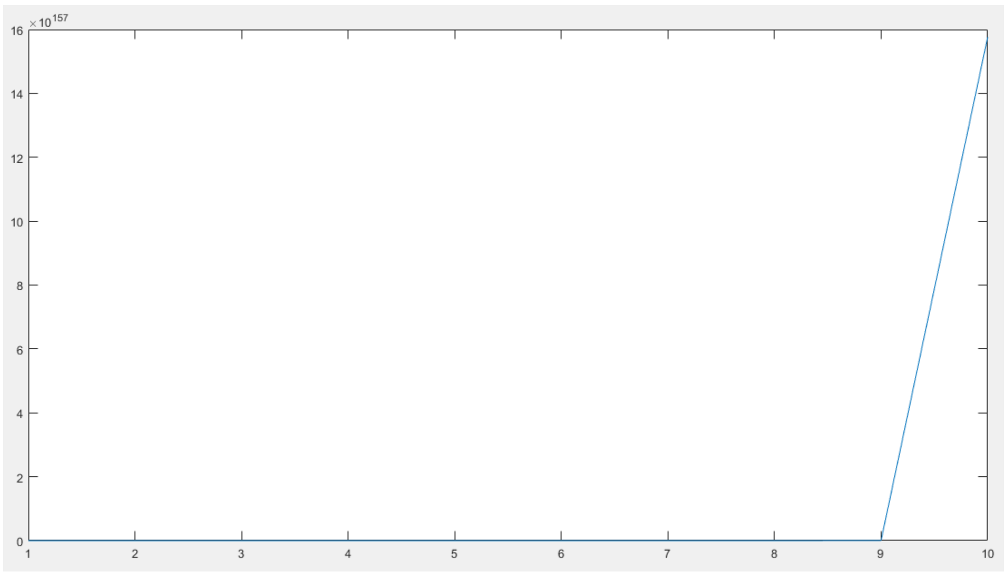
Error



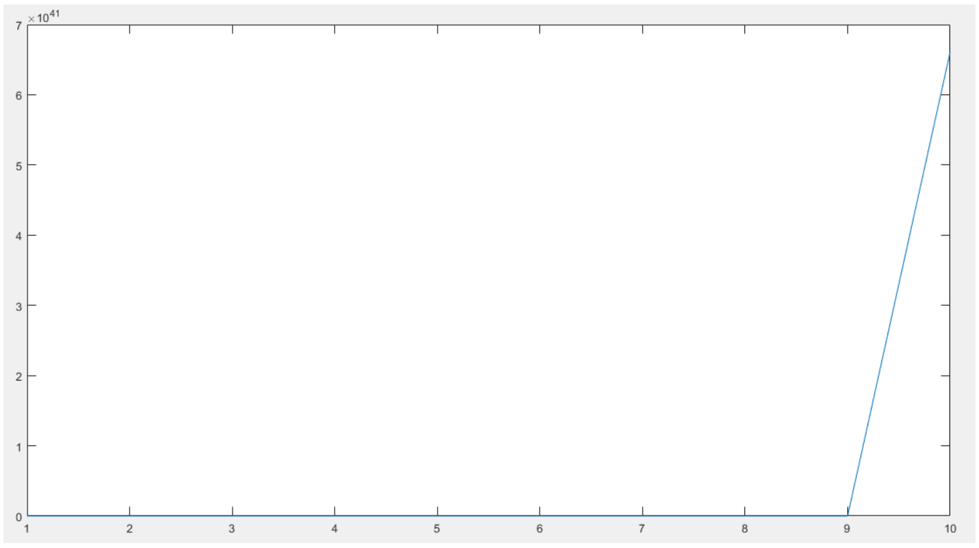
From the above plot, we see that m = 12, is a reasonable estimate.

Now, we proceed to determine weights using gradient descent. We have to choose a stopping criterion for our gradient descent, which I have chosen to be the number of iterations after which change in error becomes negligible. We also have to select the learning rate, for which we adopt a hit and trial method. For these runs, batch size is set to be 1(stochastic gradient descent), because stochastic gradient descent is the most time consuming.

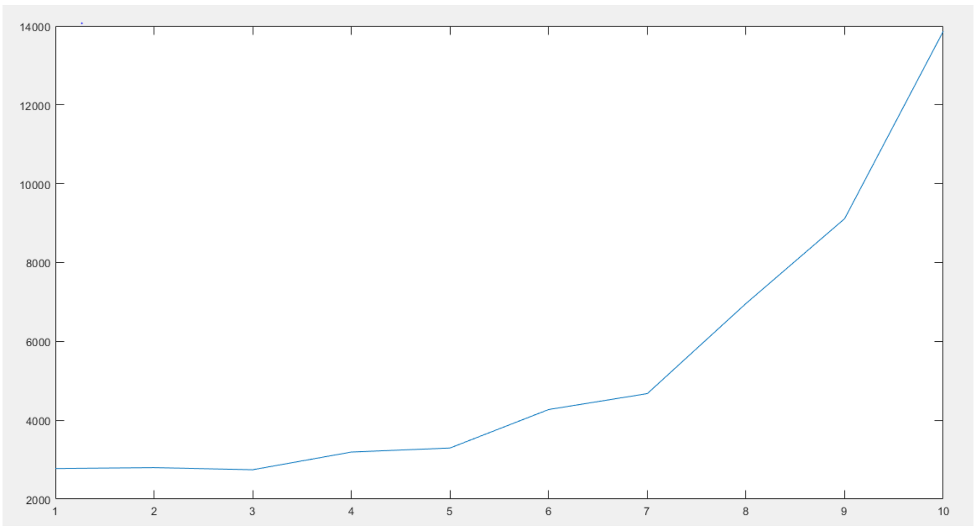
We now plot graphs of error vs iterations for various values of learning rate, starting with learning rate = 10.



As we can see, the error increases unboundedly in only 10 iterations. So now, we set our learning rate to 1 and again draw the above plot.



So now, although the magnitude of our error has decreased, but it is still increasing, so we reduce learning rate to 0.01.



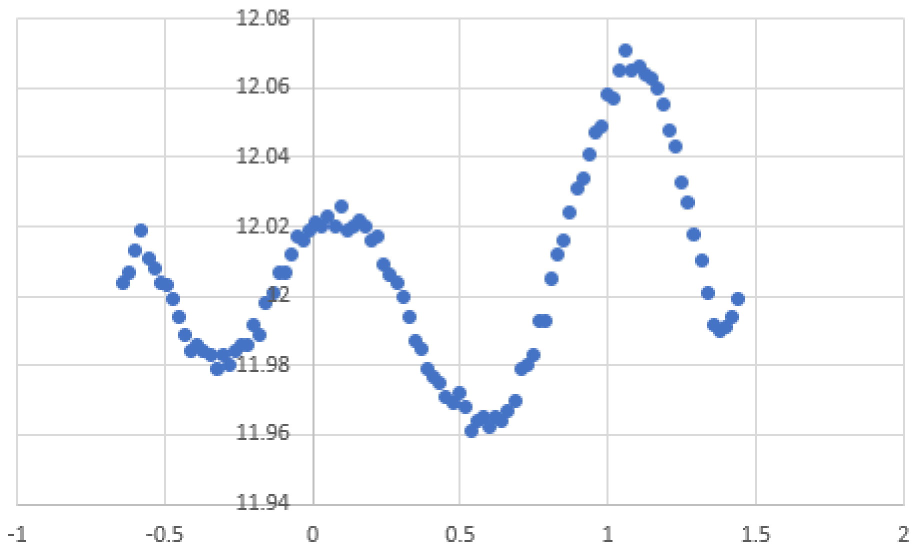
Still increasing!! I checked for 0.09 and found it to be increasing as well. Learning rate = 0.08 is the first time when the error actually decreases. Hence, it is the optimum learning rate. To find the number of iterations, I ran a few runs with different numbers and found that after 25000 iterations, our gradient descent sufficiently converges and the change in error is less than 0.001 after each iteration. Hence, finally we have our gradient descent parameters, with

Learning rate = 0.08

Number of iterations = 25000

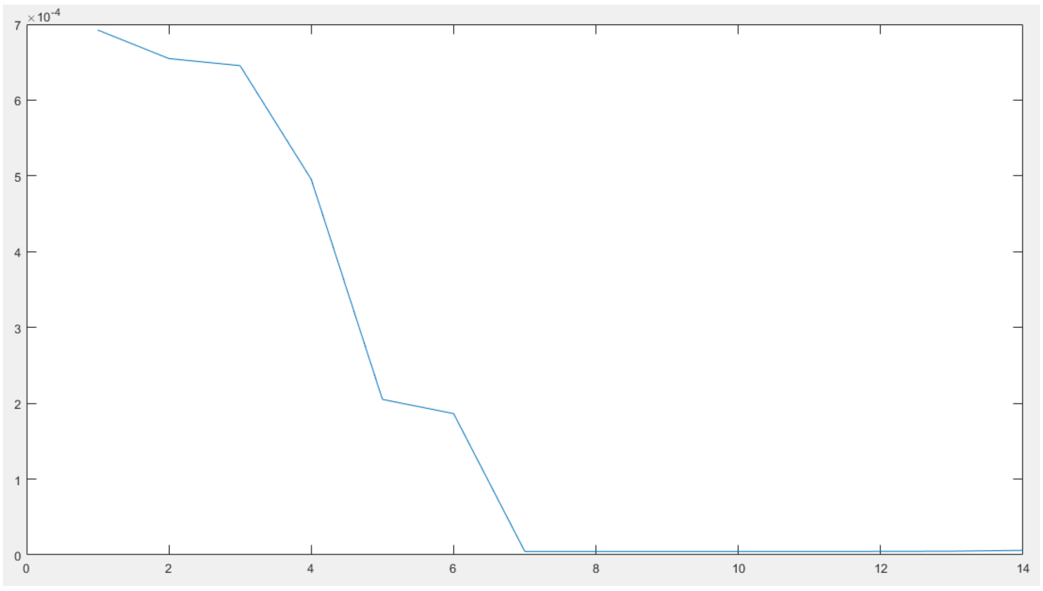
Extrapolating the above to the full data set of 100 points

Visualising the data



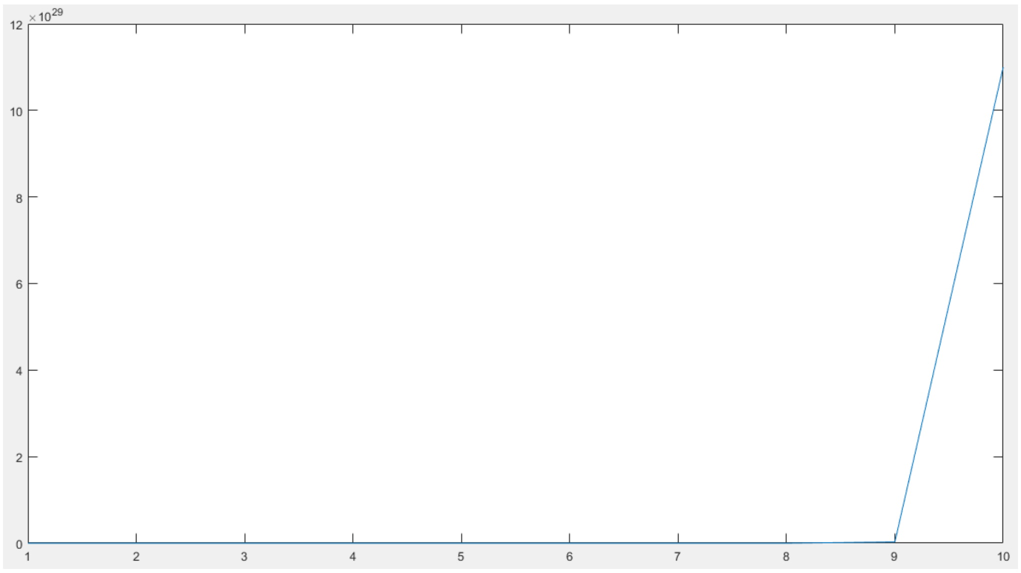
Again, as previously, we will try to get a rough estimate of m, for which we use the Moore-Penrose pseudoinverse function. We vary m from 1 to 20, create the design matrix, get the weights and the corresponding error, which is plotted below as a function of m. In this, we have used validation and split the data in a 80:20 train to test ratio.

Error



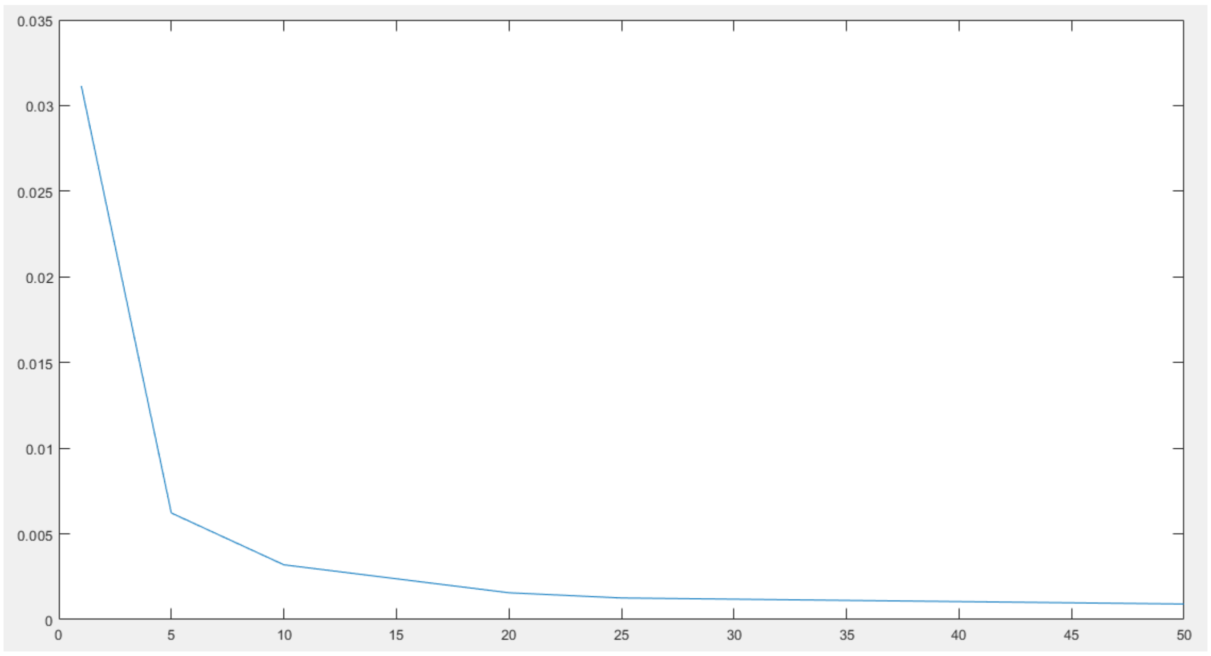
From the above plot, again m = 12 is a reasonable estimate.

Now, we try to find the parameters for gradient descent. From our previous experience, this time we start off with learning rate = 0.1. Plot of error vs number of iterations follows.



Doing a trial and error as before, we finally settle down to learning rate = 0.01 and number of iterations = 25000, which coincidentally is the same as before.

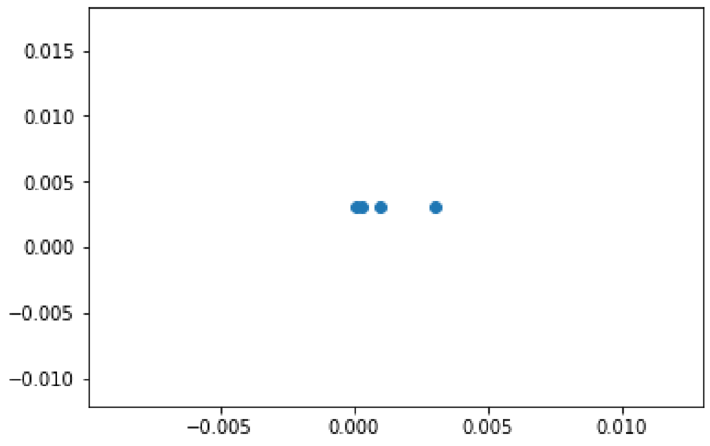
One thing which we haven't touched until now is changing of batch size in gradient descent. Above all parameters were derived for stochastic gradient descent, keeping in mind that if learning rate is low enough and number of iterations is high enough that stochastic gradient descent converges, other batches would definitely converge. Now, batch size is varied from 1 to 100 and the resulting plot for final error after gradient descent is plotted below.



This is because the number of iterations is the same for all batch sizes, and the higher the batch size, the better will it converge.

Regularisation

The first step is to choose the regularisation parameter lambda. For that, we will use cross-validation approach. In that, we do a 10-fold cross validation and the average error for each lambda in a list of lambdas is calculated. The one with the lowest average error is selected.



The above graph is plotted as average error vs lambda. The above graph is not clear due to scalability issues, but observing from the values, we get that lambda = 0.0003 is optimum lambda. Armed with this final information, we can finally provide our estimates:

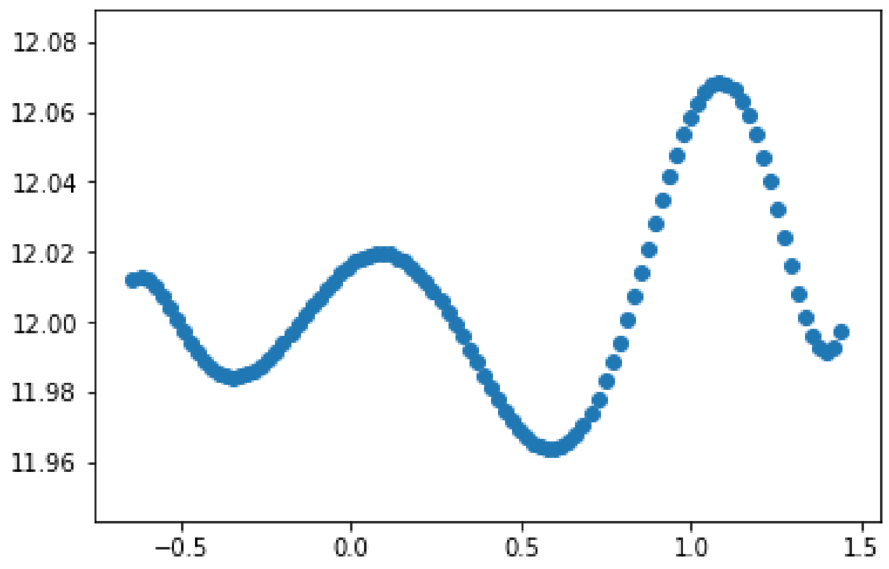
M(order of polynomial) = 12

Weights = [12.01646951],[0.07199382],[-0.35915098],[-0.58060081],[1.14413396],[0.62237857],

[-0.75953415],[0.1837603],[-0.65415219],[0.35609965],[-0.11583325],[0.22801759],

[-0.09482486]

Using the above weights to plot the predicted output, we get:

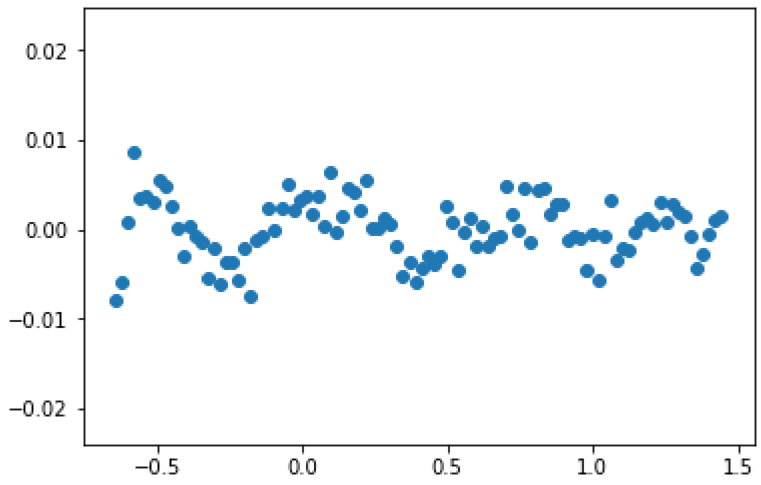


If we take a look back at the original data graph shown above, we see that the above graph sufficiently resembles the former.

Calculating the error variance

Error = y_given – y_predicted

Plotting above difference, we get:

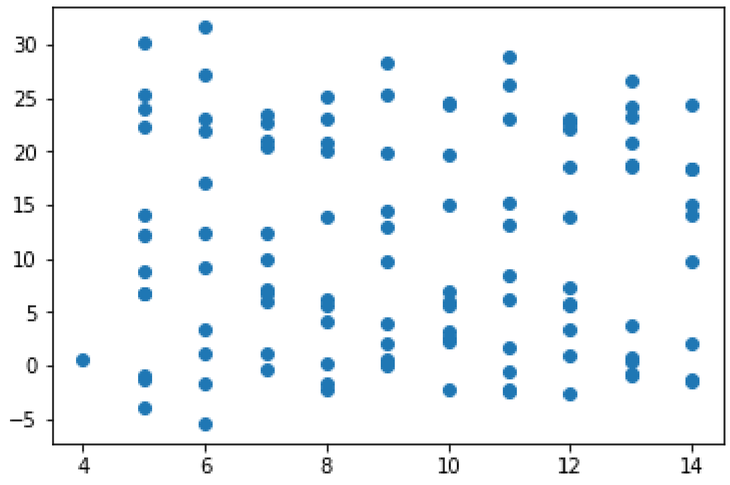


The mean of the above data is found out to be 3.6*e(-5), which is sufficiently close to zero.

The variance comes out to be 1.10866*e(-5).

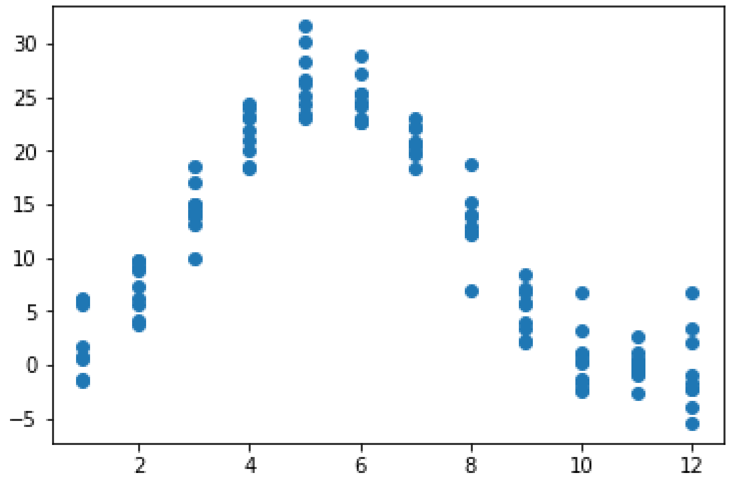
PART 2

The data has all days same, so the day can be dropped. The dependency of data on year can be seen as:

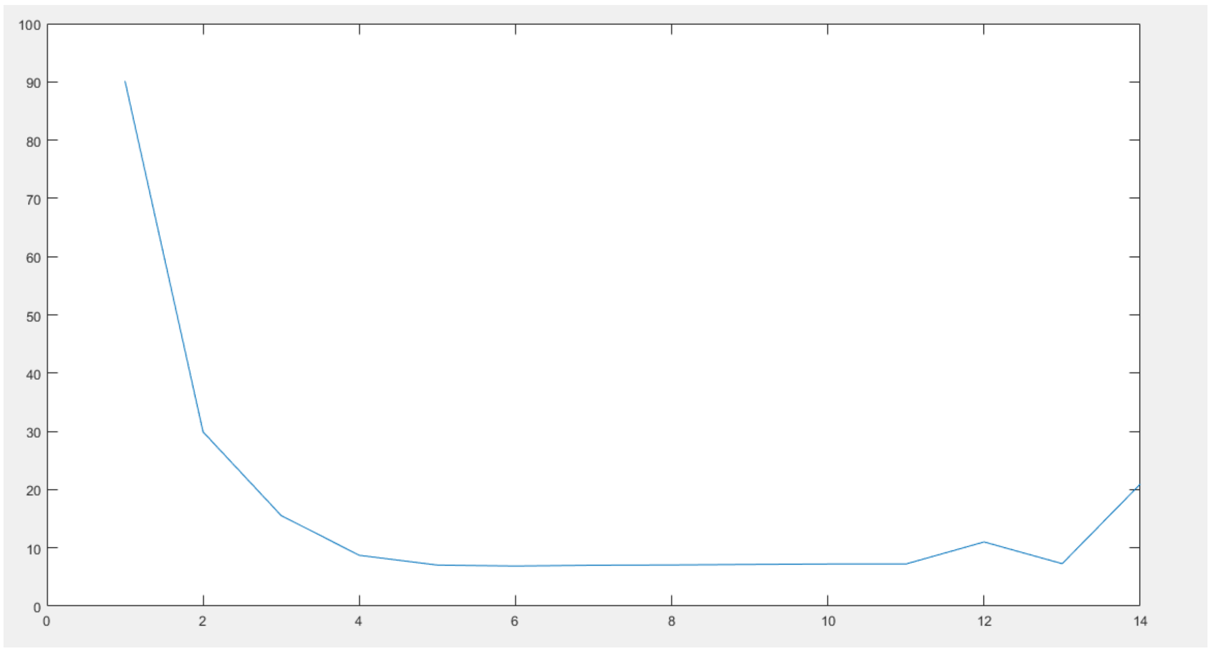


This shows a very random relation between the value and year.

The dependency of data on month:



This shows that a polynomial curve can be fit if go by month. So, we take ‘x’ as the month and try to fit a polynomial curve following our above procedure. We increase m in steps of 1, and for each m create a new design matrix (with corresponding powers of x) and do a 11-fold cross validation. The average error for each m is calculated and the same is plotted below with respect to m.



We get the lowest error at m = 6. Hence our required polynomial for polynomial regression is of degree 6. Using this, we calculate the weights using Normal Equations and predict the output for the test set.

Another insight which can be taken from the data is that data most probably represents the average temperature of the corresponding day. Temperature is seasonal with respect to the year, that is temperature in Dec 2005 will most likely follow the same trends in Dec 2006. Hence, this further justifies our exclusion of the year from calculations and using only the month as a feature.