Shilpa Kalisetty

CMSC 510: Regularization Methods for Machine Learning

Project 1, Tom Arodz

A close up of text on a white background

Description automatically generated***🡺*** *Deriving the formula for* ***g (, ) = L****:*

**=====PROCESS TO MY SOLUTION: CODE EXPLAINED IN DETAIL=====**

1. Because ‘’ values are randomly chosen using numpy’s **np.random.rand**, so my w\_old is a vector of randomly generated w values. To make sure that the SAME random values are used; I used **np.random.seed** function which makes sure it prints the same
2. Built a **9 x n** matrix, where ‘n’ is the polynomial degree:
   1. Columns represent vectors of each degree; n = 1,2,3,4,5
      1. 1st column = vector [] of just **1’s** = for (bias)
      2. 2nd column = vector [] of **x**
      3. 3rd column = vector [] of
      4. 4th column = vector [] of
      5. 5th column = vector [] of
      6. 6th column = vector [] of

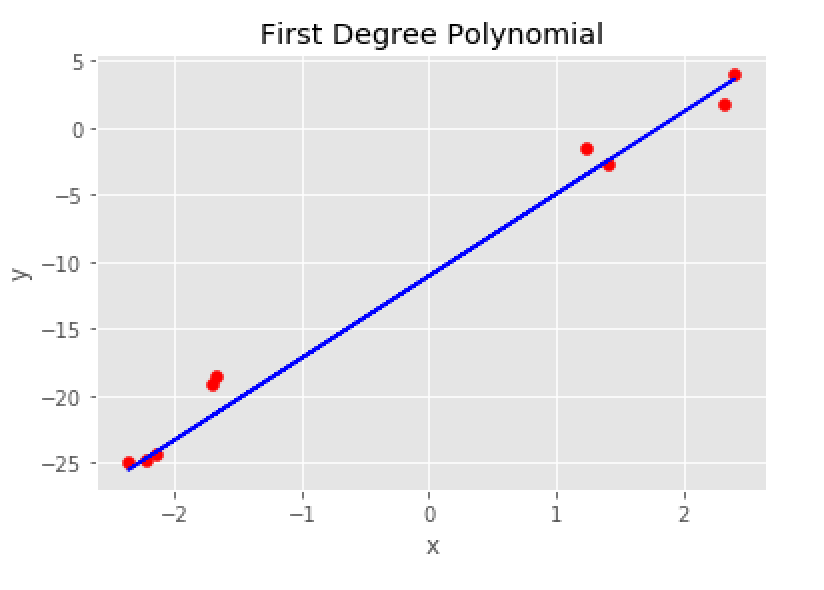
🡪 This becomes the **‘**x\_train’ vector matrix that changes size/shape whenever ‘n’ or the degree of polynomial increases/changes.

1. To calculate ‘y\_predicted’, which is what MY code/algorithm predicts the ‘y value’ to be: I perform matrix multiplication using **np.matmul** of ‘x\_train’ & ‘w\_old’ to get the resulting vector matrix of the predicted y values.
2. Then, I defined the **loss** function: L(h(x), y) = :
   1. h(x) here is the predicted Y values that I derive
   2. prints the loss function for each (x, y) pair and therefore prints 9 samples from our training data.
   3. Sums up each of these loss values
   4. Averages the sum/ dividing by the total number of samples or the size ‘m’.
   5. Derive the Mean Squared Error values
3. Final step: **gradient** descent – derivation: **g (, ) = L:**
   1. **g** = 2(() – y)\* (1 + x + )
   2. ‘g’ is calculated for every sample (x, y) and then the sum and the average is taken of the entire sample set.
4. The MSE or the squared objective function (L) is supposed to be as close to 0 as possible.
   1. I am not able to achieve this. I have tried multiple ways of using different number of hyperparameters; tuning the iterations while keeping the gamma constant and vice-versa.

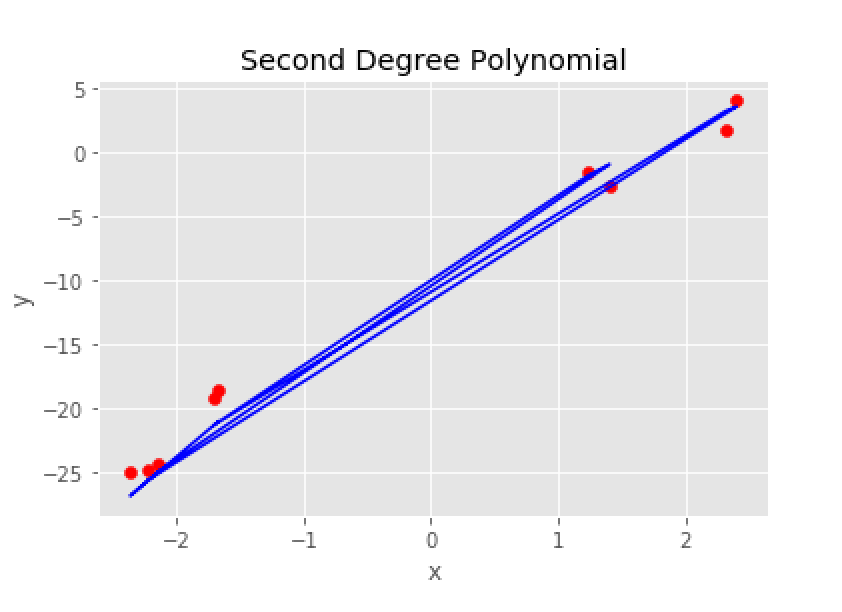
**============RESULTS=============**

* Plots for each polynomial degree ‘n’:

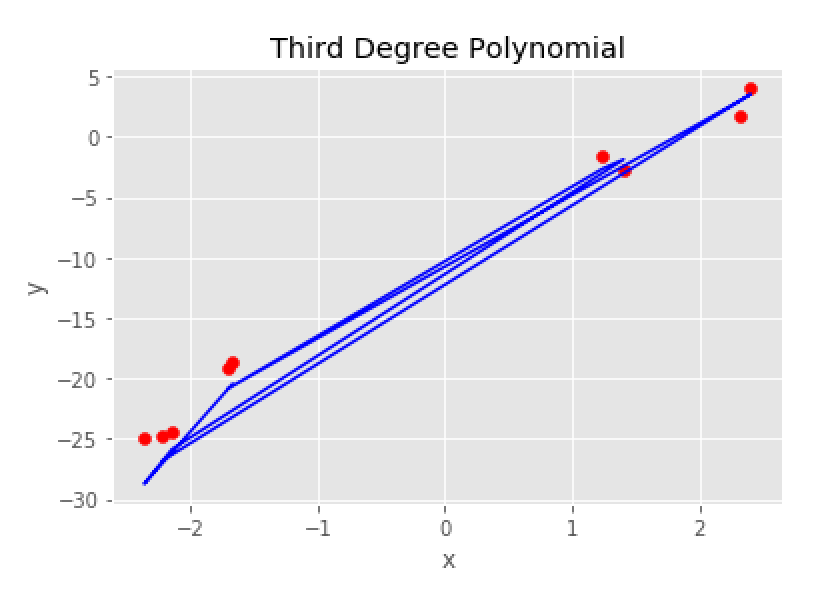
**For n = 1:**



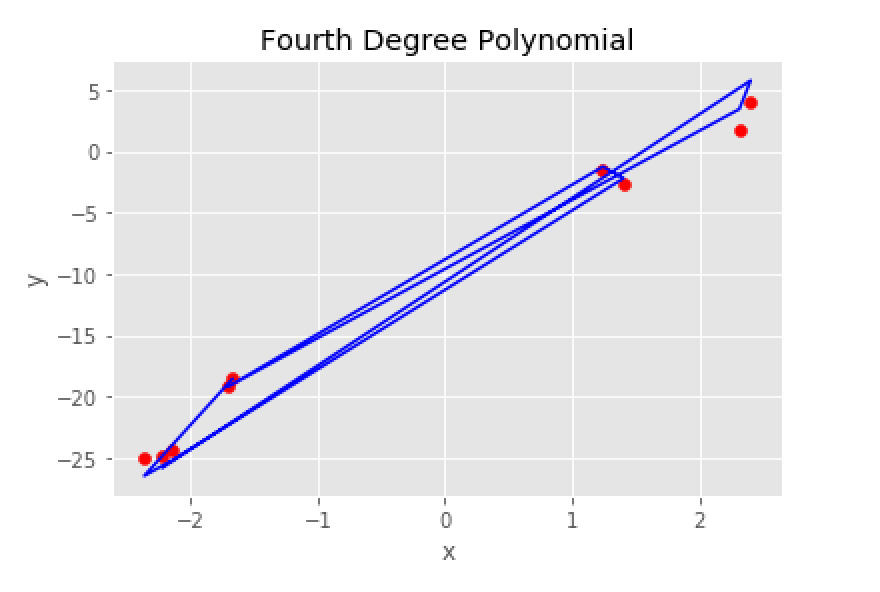
**For n = 2:**



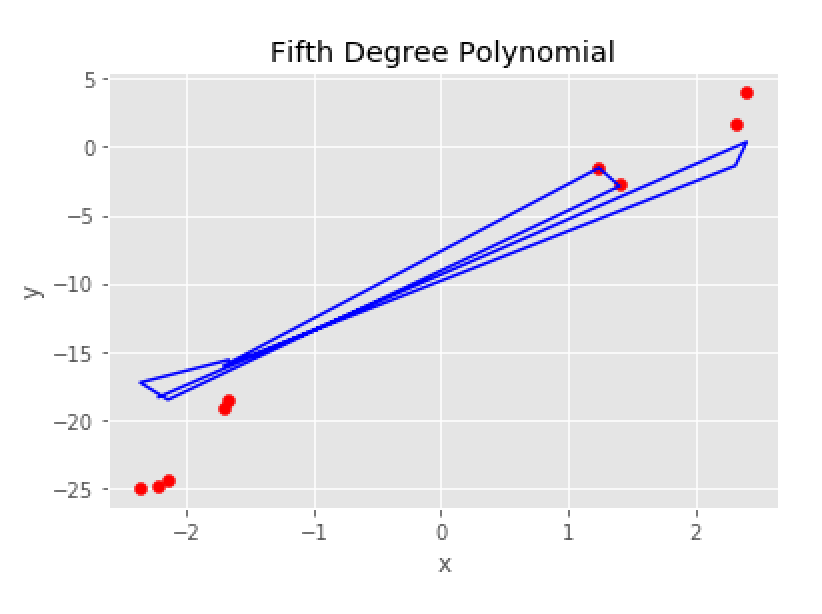
**For n = 3:**



**For n = 4:**



**For n = 5:**



**🡺 MEAN SQUARED ERROR:**

MSE was computed by first taking the sum of the differences between the predicted and the actual ‘Y’ values squared and then dividing that by the total number of samples (training data).

The learning rate and the number of iterations are also specified in the code; to improve and reach the ‘good’ values of w.

The goal here is to achieve optimized ‘w’ values such that the cost function is as minimum as possible.

* This is where I was having problems. I was not able to achieve a minimum cost/ MSE error rate and I could not figure out why.

|  |  |  |  |
| --- | --- | --- | --- |
| **Polynomial Degree (n)** | **MSE** | **Gamma** | **Number of iterations** |
| n = 1 | 121.432 | 0.001 | 300,000 |
| n = 2 | 103.856 | 0.001 | 100,000 |
| n = 3 | 83.951 | 0.0001 | 30,000 |
| n = 4 | 67.808 | 0.001 | 15,000 |
| n = 5 | 39.312 | 0.00001 | 50,000 |

It was extremely time-consuming and difficult to get the best MSE values for both the polynomial degree for n = 1 and n = 2. The performance of these two degrees were particularly very slow as tuning any of the hyperparameters wouldn’t produce the best results.

Another confusion was: I was getting ‘Y Prediction’ values to be very close to ‘Y actual’ values => h(x) = y but my MSE wouldn’t do better; which was not making sense to me.