Linear Regression Write-up
Homework #3
By: Alexander Polus
For: Dr. Brent Harrison
3.3.18

Introduction

For homework 3, the class was asked to build a linear regression model that could train against the three provided synthetic data sets, generate a set of theta values for 1st, 2nd, 4th, and 9th order polynomials. Below I will walk the reader through my code, and why I implemented this algorithm the way I did.

Code Summary

First, I opened the data as a csv file, and turned the data into a 2D list. The outer list forms the rows of the table, and each inner list is length two, consisting of the X and Y values from each row in the csv file corresponding to indices 0 and 1, respectively, in the inner lists. Values were converted to Floats.

Next, I established my alpha, or learning rate. I chose a value of 0.0001 because I didn't want my model to "learn" too fast and diverge, given the relatively small range of values given. I elected to pass up normalizing or standardizing my input values, because I wanted to maintain the visual shapes of the data for graphing purposes. After that, a VectorMultiply() function was defined but never used – Numpy's inner dot product function proved much more effective.

I defined a set of functions that would turn one input value, x, into a list of x^0, x^1, … , x^n, where n equals the order of polynomial desired. This would later be used to form a dot product with my theta values.

Although it may seem somewhat redundant, I was unsure about passing functions into arguments of other functions, so I defined a function for each order polynomial to determine the corresponding values for theta. The logic for all four functions is equivalent, and much of the code identical. For each order polynomial, I generate a set of default theta values, which I initialized all to zero. Next, I establish a counter to keep track of how many times I've gone through the entire dataset. Because my learning rate was 0.0001, I chose to pass through the data 1,000 times, and have that be my stopping criteria for learning. I understand in practice it would be better to stop when my theta values stop significantly changing or when the error crosses a certain threshold, but that seemed unnecessary for this data.

The only function that was really different in any way was the function for the 9th order theta values, which differed because I multiplied the learning rate by 0.1. I did this because I kept receiving "nan" for my theta values in a couple of the datasets, which I interpreted as diversion, and lowered the learning rate so my model would converge. I understand in practice that it would be better to have an adaptive learning rate, but the 9th order theta function was the only part of the model that diverged, so I didn't want to mess with any of the other code. Theta values in each case were determined using Stochastic Gradient Descent.

For my cost function, I used mean squared error, which I calculated by finding the total squared error across all data points, then dividing that by the number of data points.

The "main" section of my program finds all theta values for all four polynomials, and the error, and prints all of those values out.

Functionality, Usage, Preconditions and Postconditions

Functionality: This program, upon execution, will print out the training dataset row by row as an array of float values, will then indicate that it is running (although it says 20 seconds max, will probably take around 5), then it will print out the theta values, in order from theta sub zero to theta sub n (where n is the order of polynomial) for each model along with its mean squared error.

Usage: python3 LinearRegression.py <filename>. Data must be present in the same folder as the executable.

Preconditions: A file with two columns, X and Y, which correspond to data points

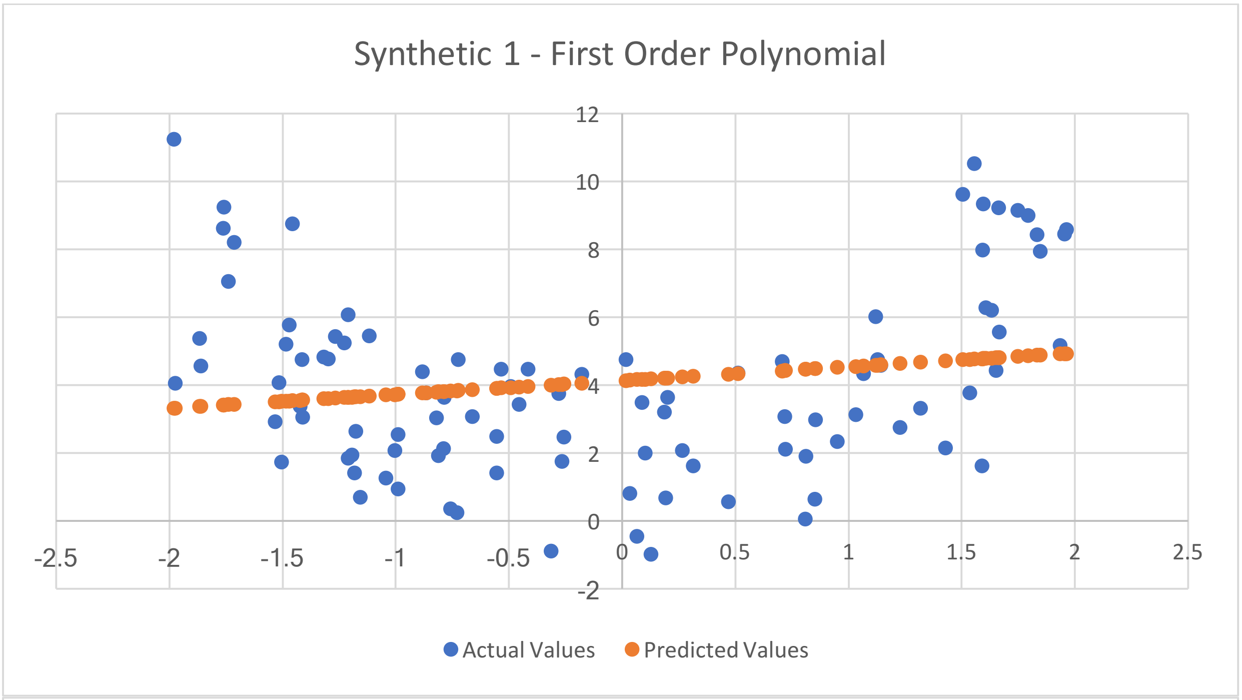
Postconditions: the training data as floats, the theta values for each polynomial, and the error for each model.

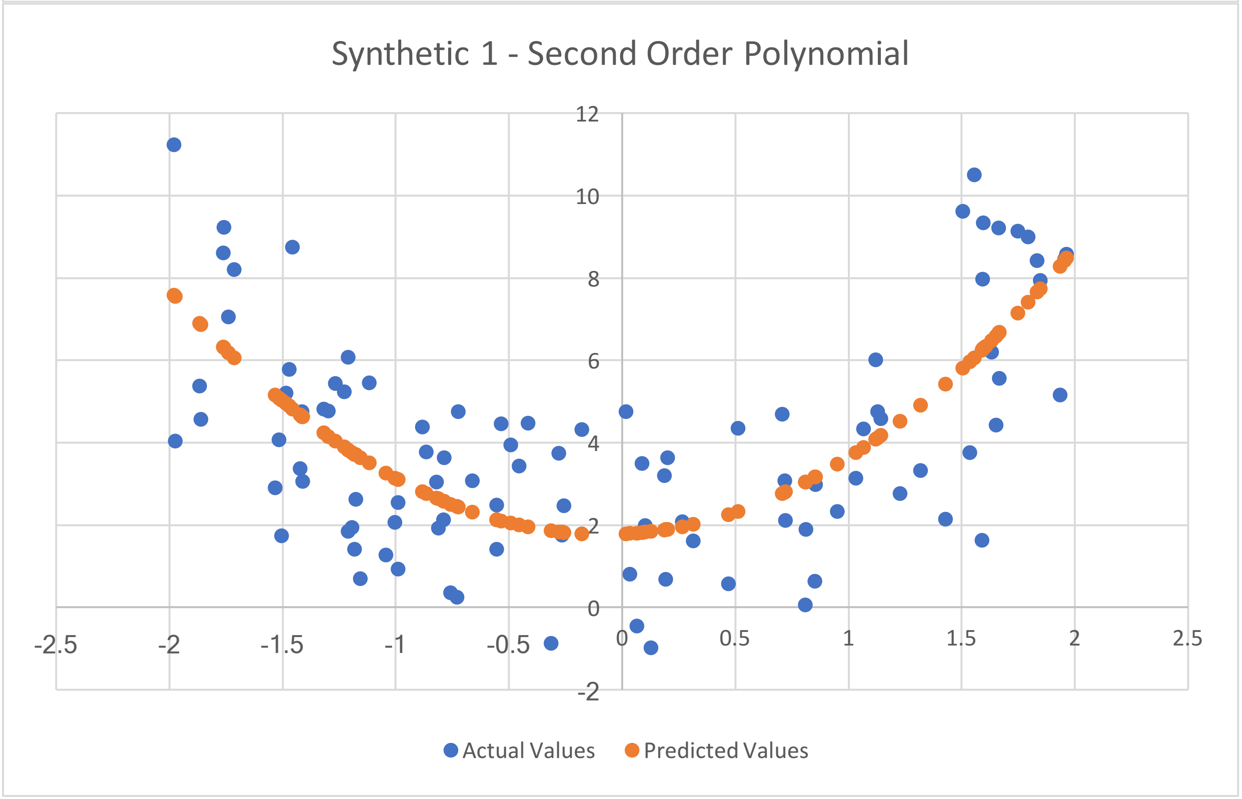
Conclusion

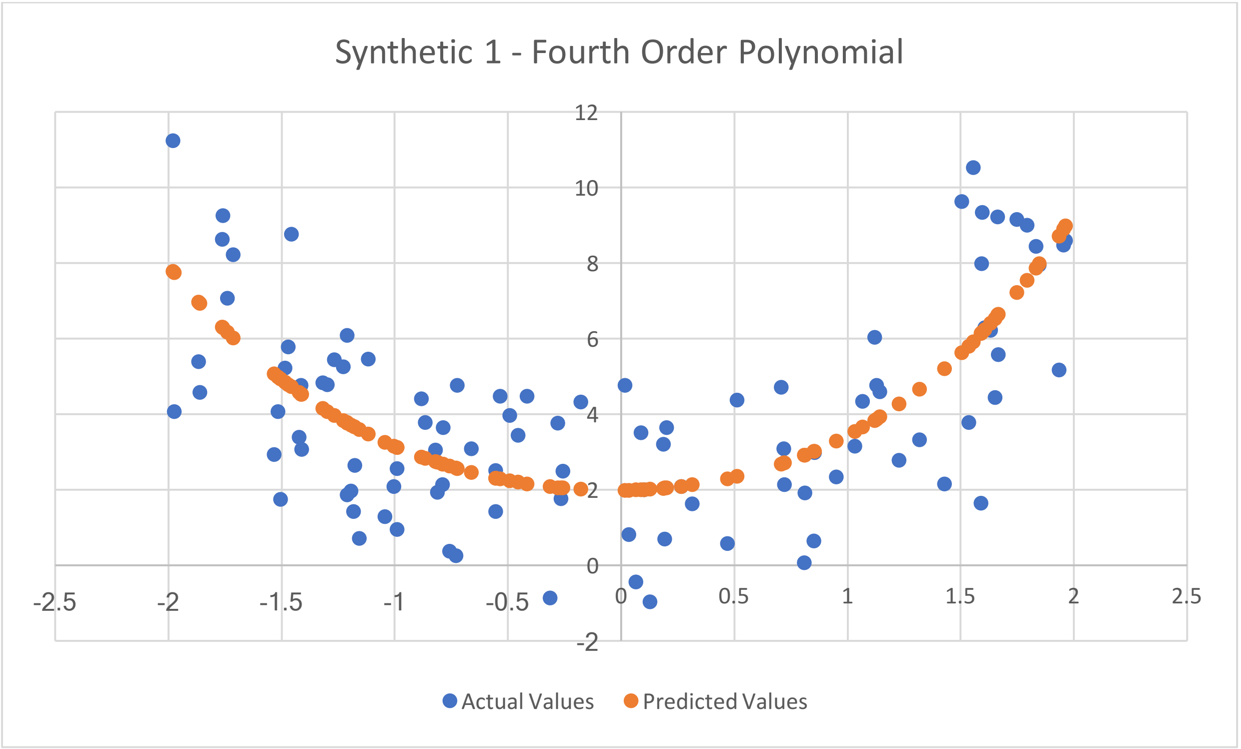
The results of each model are illustrated in the 12 graphs in the appendix. I did my visual processing in Excel, the results of which can be found in this submission. For each model, I list the X and Y values, then the Hypothesis value. Each graph shows the actual values and predicted values in each model. I elected to go point by point instead of drawing smooth lines; for me, this method was clearer. The errors of each model are in the table at the bottom of the appendix. The Fourth order model was the most effective for modeling each dataset.

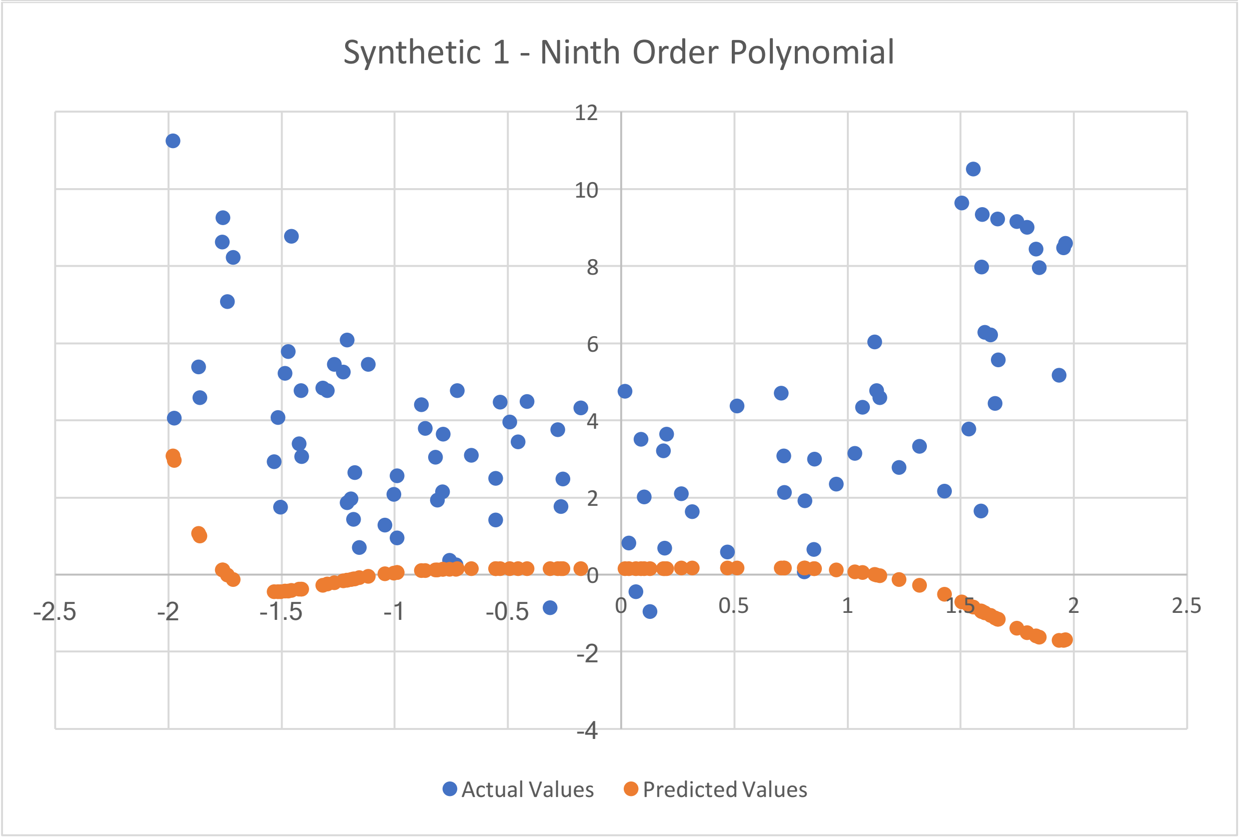
Appendix

Synthetic 1 Results

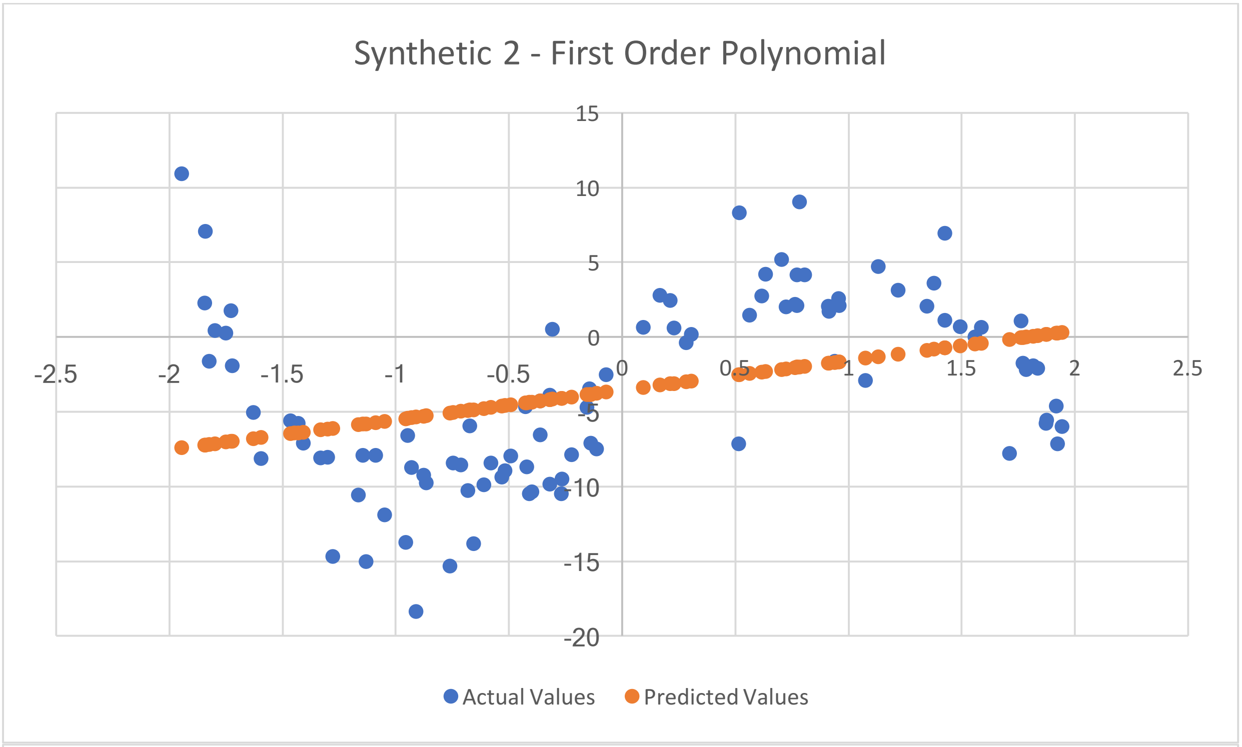


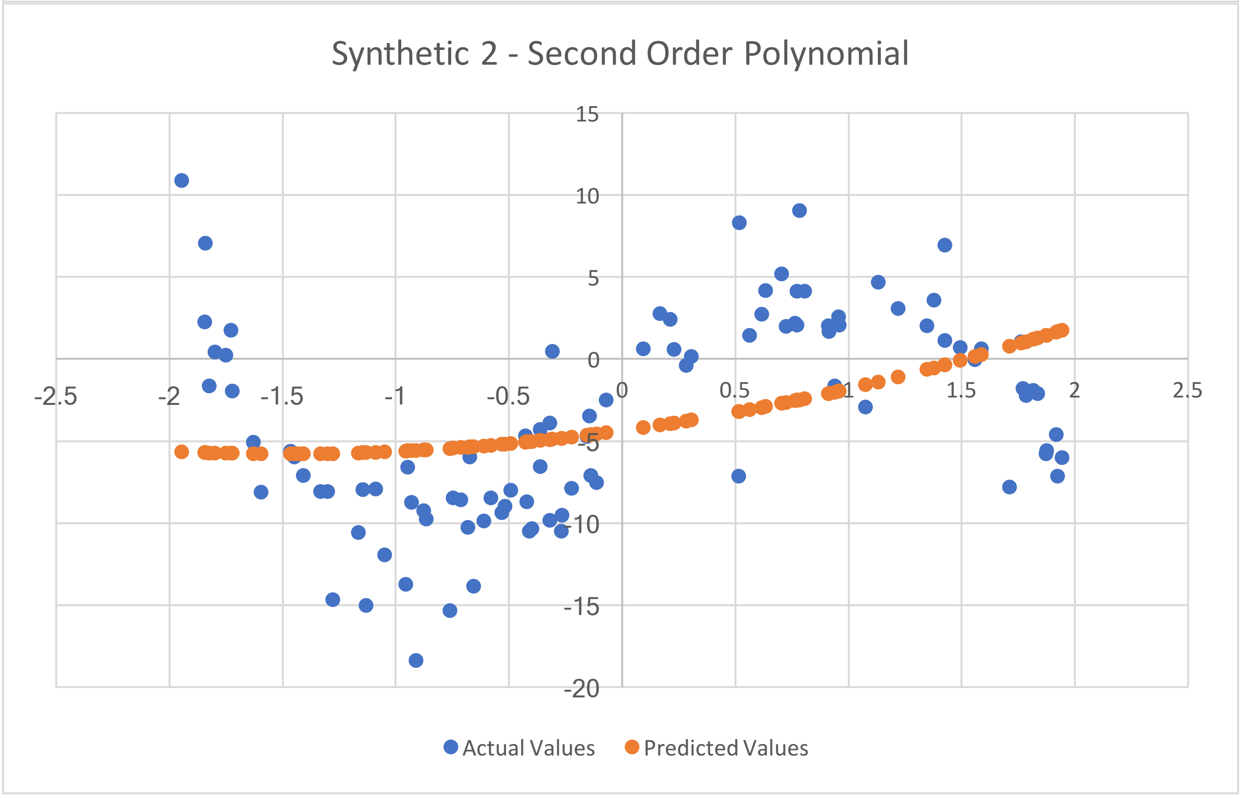


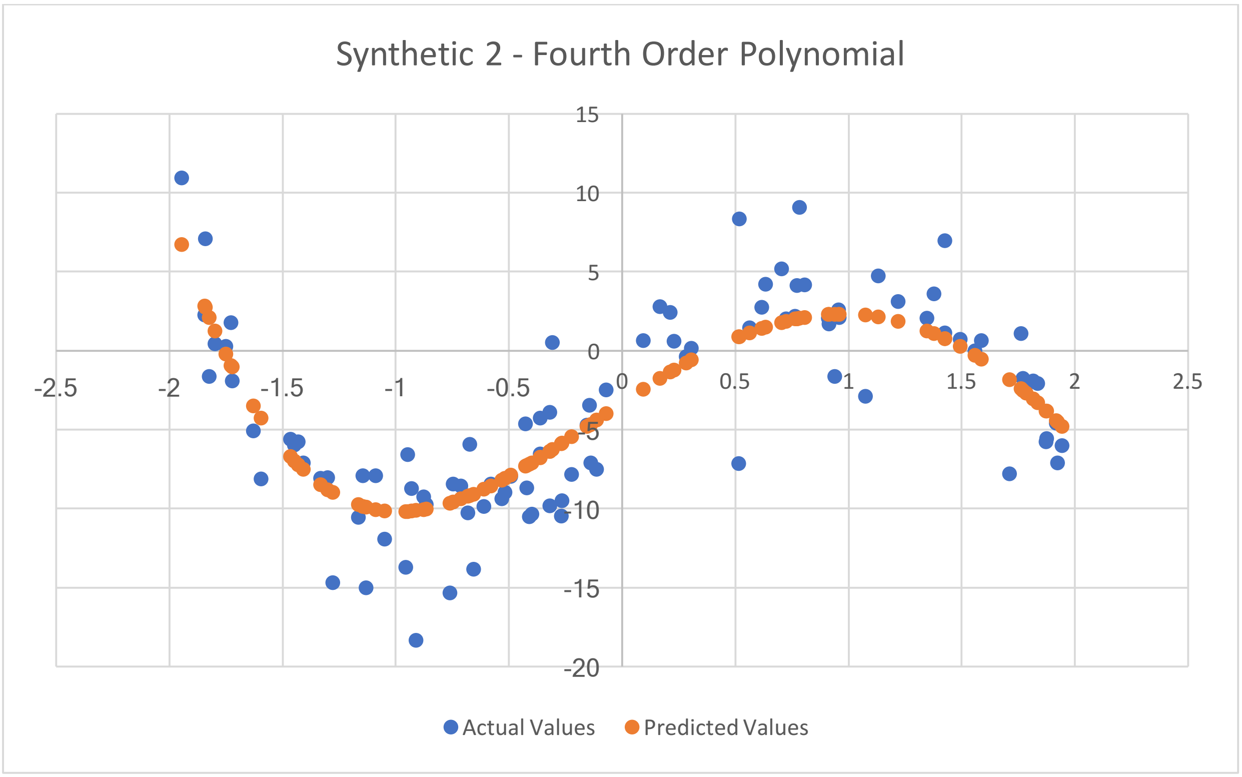

Synthetic 1 - Fourth Order Polynomial
12
10
8
6
4
2
0
-2
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values

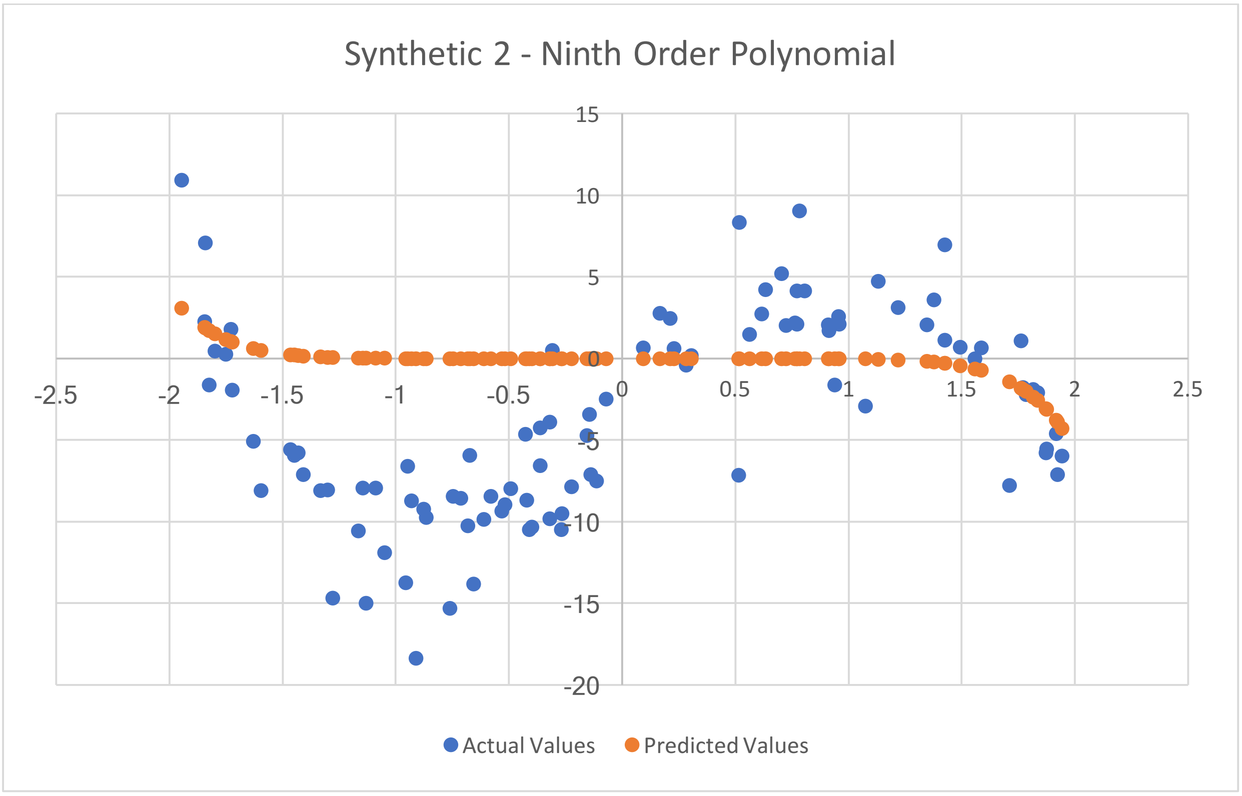

Synthetic 1 - Ninth Order Polynomial
12
10
8
6
4
2
0
-2
-4
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values

Synthetic 2 Results

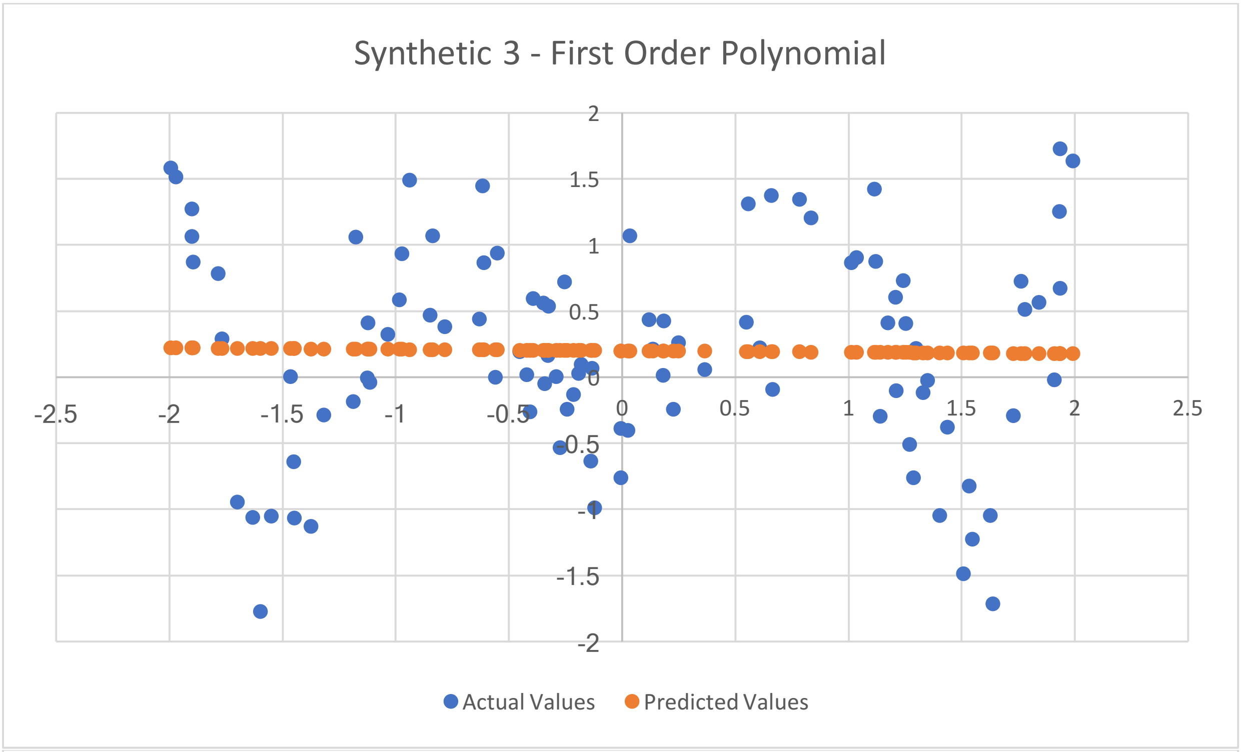


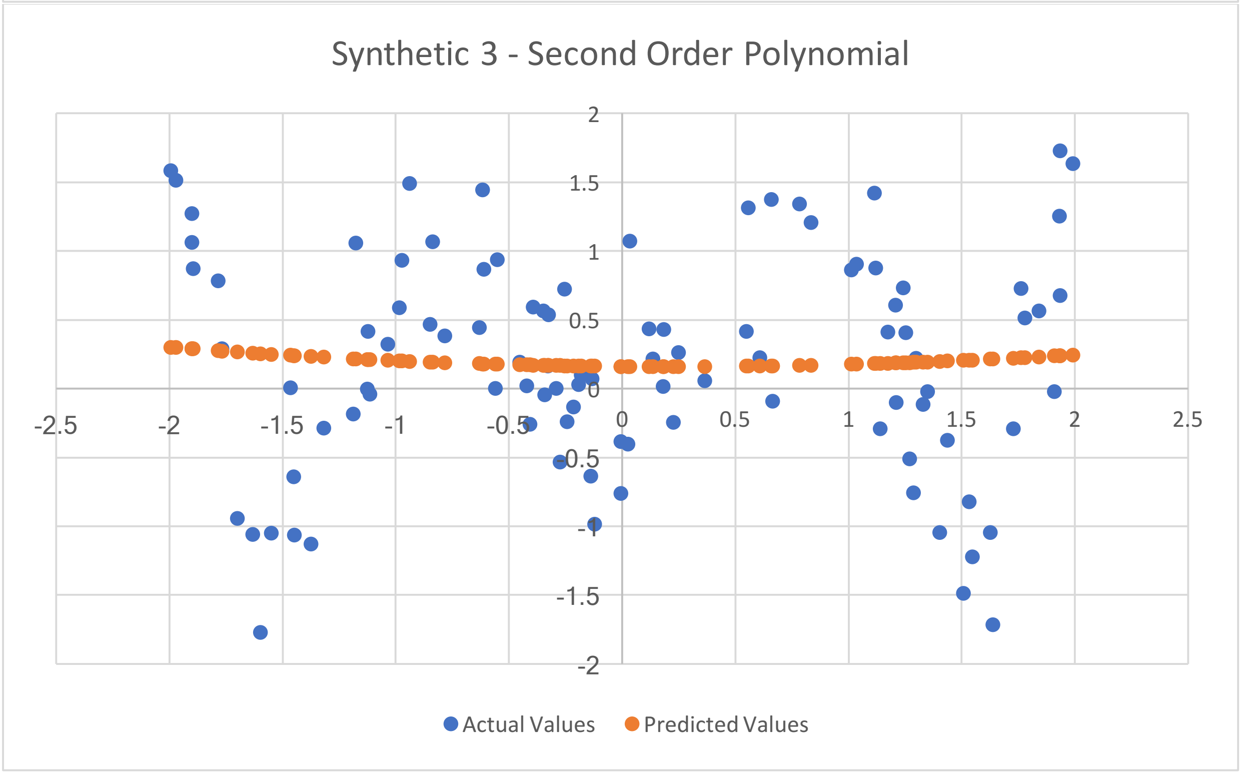


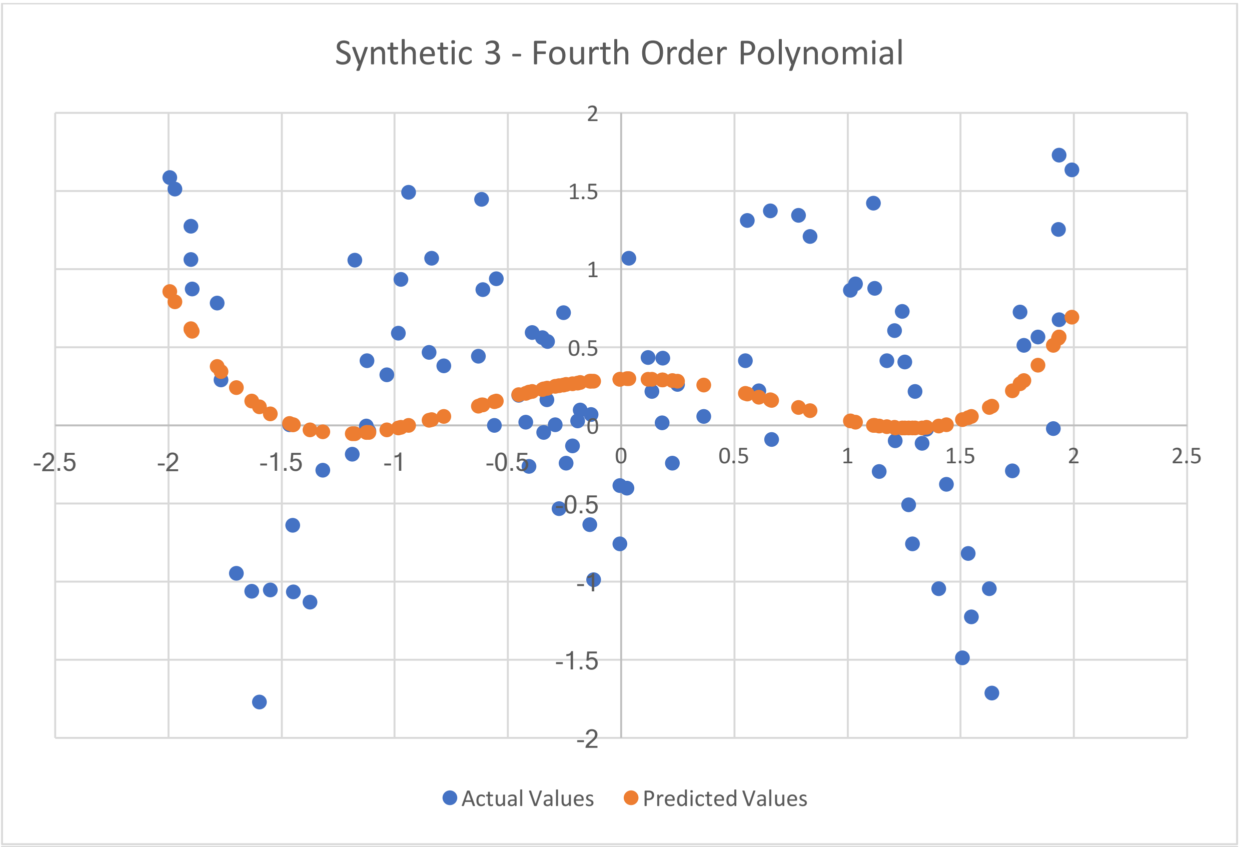

Synthetic 2 - Fourth Order Polynomial
15
10
5
0
-5
-10
-15
-20
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values


Synthetic 2 - Ninth Order Polynomial
15
10
5
0
-5
-10
-15
-20
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values

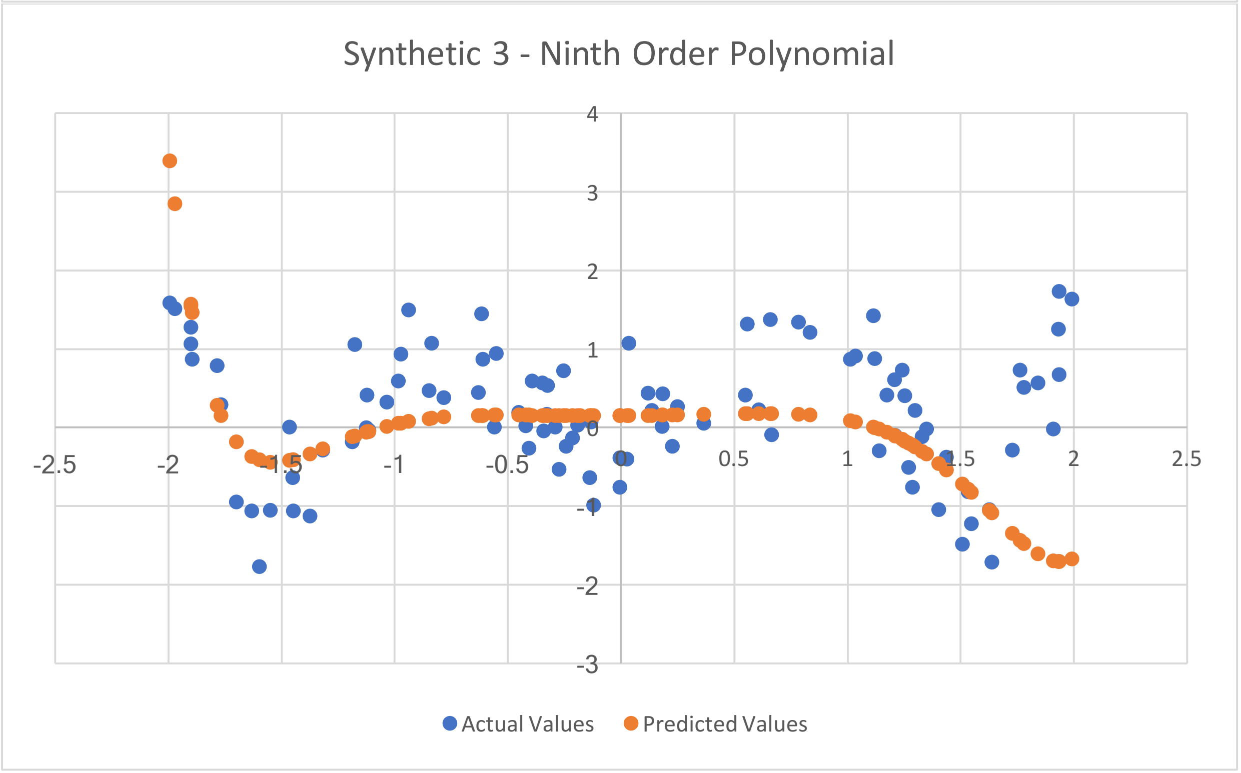
Synthetic 3 Results







Synthetic 3 - Fourth Order Polynomial
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values



Synthetic 3 - Ninth Order Polynomial
4
3
2
1
0
-1
-2
-3
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
● Actual Values
● Predicted Values

Error of Each Model	Polynomial Order	Error
Synthetic 1	First	3.5906248785982275
	Second	1.8855586701719111
	Fourth	1.8737480454394733
	Ninth	5.813195654722423
Synthetic 2	First	15.530969124478352
	Second	15.20410361749758
	Fourth	4.454257575139275
	Ninth	9.982500919865029
Synthetic 3	First	0.3133560539099557
	Second	0.3127913254242538
	Fourth	0.27980075583728614
	Ninth	0.46896871353613556