Algorithms for Data Mining – Assessment Item 1, Task 1

Polynomial Regression

# Section 1: Description of Polynomial Regression

## Error function used for regression

An error/cost function in regression is essential as it displays the difference between the ground truth and the predicted values, the most commonly used function used is the (Root) Mean Squared Error. The function calculates the error/variance for each point in the dataset from the best fitting line, squares it and then calculates the mean from all those points. The reason for squaring the error value is to ensure that the value is positive (as the variance can be negative), and will highlight outliers. With MSE and RMSE, the lower the value the better, and if you were to have a perfect best fitting line, the value of the MSE/RMSE would be 0. Rooting the MSE enables the values to be more easily read, as it will give you the distance on average from the best fitting line across all the values in the dataset.

The mean squared error cost function can be expressed mathematically, with the RMSE being a simple root of the formula below:

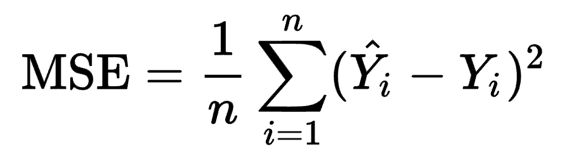


Figure 1 - Mean Squared Error Cost Function

The formula subtracts the predicted individual Y values from the ground truth Y values, before squaring them. From there, the mean is calculated from those values. That would therefore produce the MSE, and to produce the RMSE you could simply square root the MSE.

An easier to read Python solution is below:

Figure 2 - Python solution for Root Mean Squared Error

error = y\_pred – y

squaredError = error \*\* 2

meanSquaredError = squaredError.mean()

rootMeanSquaredError = np.sqrt(meanSquaredError)

Which can be simplified to:

Figure 3 - Simplified Python solution for RMSE

rmse = np.sqrt(((y\_pred - y) \*\* 2).mean())

## Linear Regression Models

## Least Squares Solution

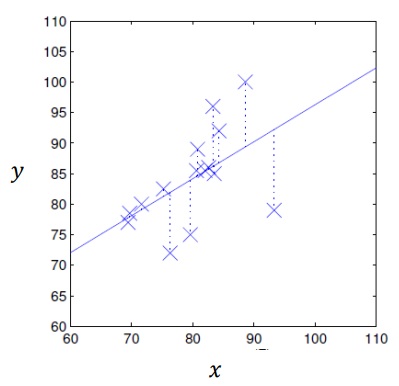
The concept for the ordinary least squares method is to calculate and reduce the sum of the squared errors produced by each point at how far it is away from the line. The error in this case is also sometimes referred to as the ‘residual’. Looking at *Figure 4*, showing the ordinary least squares method in action on a linear regression line, the points (marked with X’s) that are close to the line will have a very low error value, whereas the outliers on the graph (the points far below and above the line) will have a large error. Calculating the lowest sum of the errors produced by the points will result in the ‘best fitting line’.

Figure 4 - Error cost in ordinary least squares

…. Add equation

## Polynomial feature expansion

Polynomial feature expansion and its use in polynomial regression is the main step between simple linear regression using least squares and polynomial regression. Often also referred to as a Vandermonde Matrix, the process of expanding the features of values to *n* degree allows for multiple polynomials to create the best fitting line for the degree chosen.

The process to expand the features is as follows:

Taking a degree *n*, and an array of values *X* you need to loop through from 0 to *n* raising variable X to the power of *n* each time. The resulting shape of the array will be *j x n*, this can be expressed visually as:

In practice, the first column of polynomial feature expansion will always be 1, as being multiplied by zero is 1. An implementation in python could be expressed as:

The library *numpy* offers a function *vander* to perform this operation for you, which would be implemented as, giving the same result:

Figure 5 - Polynomial Feature expansion (Vandermonde Matrix) implementation in Python

X = np.vander(x\_train, degree -1)

Figure 6 - Simplified Python implementation using libraries

# Initialise 2d array of size x\_train by degree

X = np.empty((len(x\_train), degree))

# Add the nth degree feature expansion

for i in range(0, len(x\_train)):

x = x\_train[i]

for j in range(0, degree):

X[i, j] = x \*\* (j + 1)

# Flip the array from back to front and add 1 to beginning of X

X = np.fliplr(np.c\_[np.ones([len(x\_train), 1]), X])

This code will be explained later in this report, and is simply for reference in explaining Vandermonde Matrices/Polynomial Feature Expansion.

## How to use polynomial features in linear regression

Where simple linear regression and polynomial regression use the same least squares solution as shown in *Figure …,* They differ in that the X values for the polynomial regression are expressed as a matrix of with *n* degree. Both methods can produce the same results in certain circumstances, if you were to enter a degree of 1 into a Polynomial Regression algorithm using least squares, it would in theory produce the same result as simple linear regression using least squares (a straight linear best fitting line).

To use polynomial features in linear regression you will need to simply substitute the 1d array of the X values for a feature expanded Vandermonde matrix previously created into the equation *Figure …* to produce the polynomial parameters for a best fitting line to the degree.

## Difference between training set and test set performance

In most cases in linear and polynomial regression, performance is measured in the RMSE. This gives an indication of how close all the points are to the theoretical best fitting line. The lower the number the better, however a very low RMSE on the training set of data could give an indication that the data has been overfitted.

Overfitting is the act in Polynomial Regression of raising the degree of expansion too high, causing the best fitting in the training set to follow every point closely, and not taking a general line over where the points lie. This therefore means that in the test set of data, the RMSE may well be much higher, as the points are not in the same places as in the training set. Looking at *Figure 7*, You can see that the best fitting line for the Overfitted graph follows each point too closely, and would in that case have a close to zero RMSE. However, the Good Fit/Robust graph would produce a better and more accurate result for the test set of data, as it follows the general course of the points on the graph.

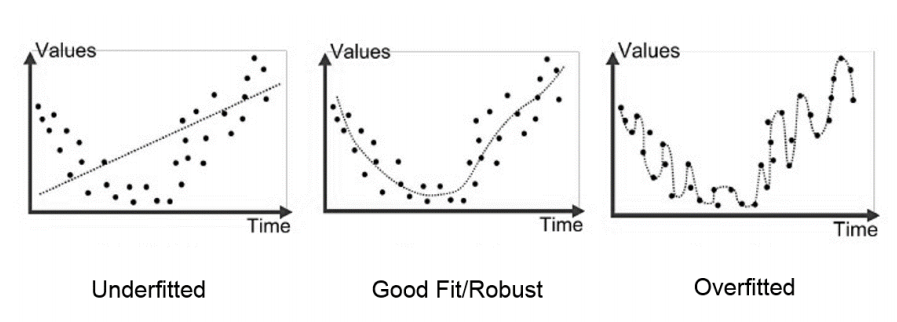


Figure 7 - Line fitting for polynomial regression

# Section 2: Implementation of Polynomial Regression

My implementation of polynomial regression attempts to avoid major library usage, and implements feature expansion manually between lines 10-15, as well as using the actual least squares solution on line 18.

Looking first at the polynomial feature expansion to create a Vandermonde matrix, it begins on line 10 creating an empty 2d numpy array (matrix) to the size of x\_train by the degree + 1 (As the degrees go from zero to degree). A for loop is then implemented, going through each of the x\_train values, and initialising the iter variable. The iter variable zips together two ranges, one going from zero to the degree + 1 and the other going the opposite way. This is required for arranging the values correctly for future use in the least squares solution, as the first variable in the matrix for each x\_train value needs to be 1 (as it is x\_train to the power of 0)

Moving to line 14 and 15, where an additional for loop goes through each degree from zero to the degree using the iter variable. For example, in a degree of 5, the first iteration of that for loop would give values j = 0, k = 5 allowing for a correct order in the matrix. The resulting variable on line 15 X[i, k] raises the x\_train value to the power of j, allowing for the feature expansion. This process is repeated each time until the degree is reached.

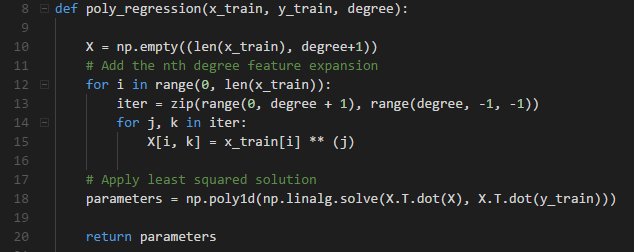


Figure 8 - Polynomial Regression Implementation

With the Vandermonde matrix populated, the function can move onto the least squares solution on line 18. This implements the equation as previously discussed in Section 1 – Least squares solution, using the X matrix with the feature expansion. In order to control matrix inversions, the numpy function np.linalg.solve is used as it offers a more precise result over using np.linalg.inv. Returning from this function, it is encased in a 1d numpy matrix for later use as parameters when predicting values. This is then returned on line 20, completing the function and producing the output parameters to the degree selected.

Below shows *Figure 9* with the Training results for each degree plotted:

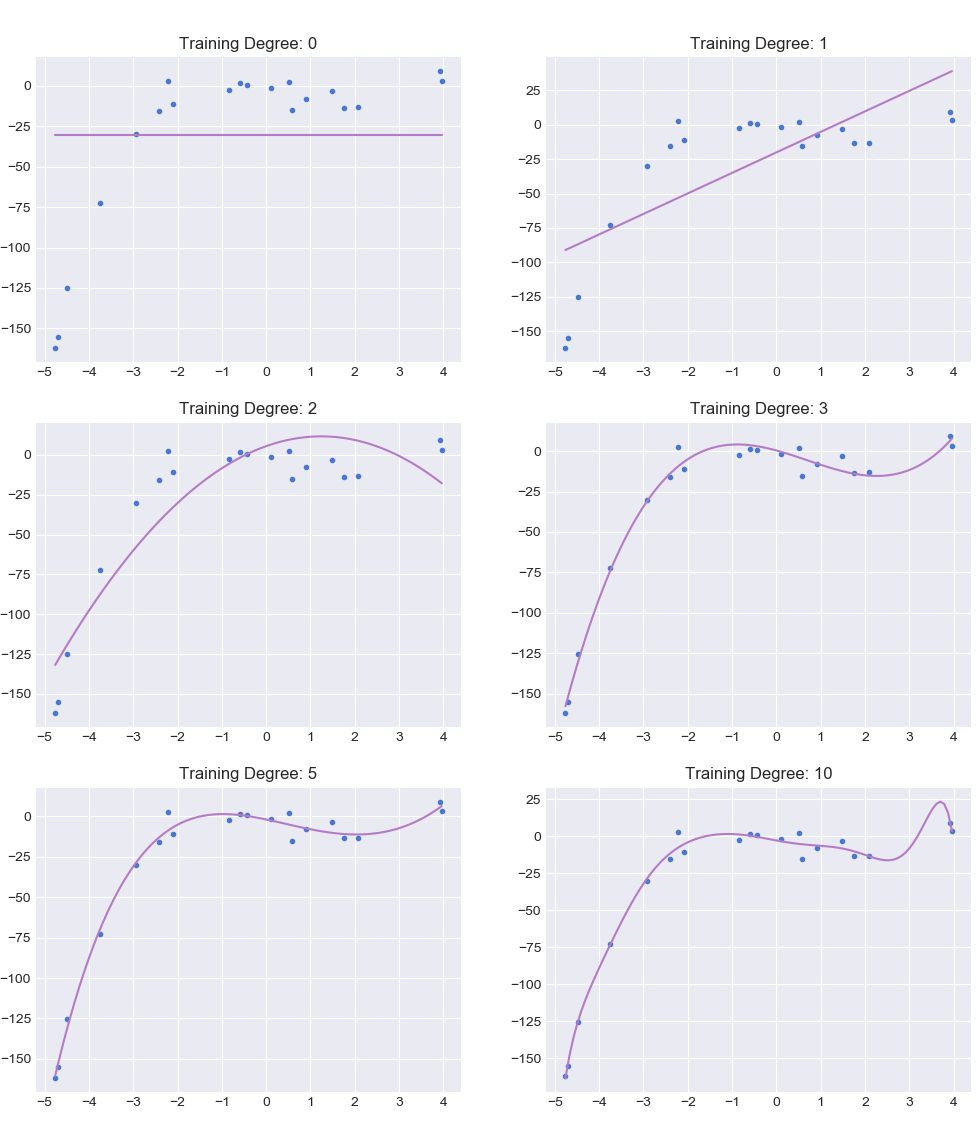


Figure 9 - Training Polynomials

# Section 3: Evaluation