Symbols, Patterns and Signals

Multivariate Polynomial Regression on 2D data using Least Squares

William Parker gg18045

14th March 2020

1 Methods and theory

1.1 The model

The aim of this program is to find a function which best fits a set of unknown two-dimensional data-points. The data points are are initially stored in a (2,n) matrix. I chose to use multivariate linear/polynomial regression to get my maximum likelihood estimator and a deterministic model to fit the data to one of a set of functions (choosing the smallest error value produced by the sum squared error of the returned maximum likelihood estimator values).

The program initially reads in the files containing the data and splits the larger arrays containing the whole data set into arrays of length 20; two arrays $\mathbf{X}$ and $\mathbf{Y}$ are produced for each set of 20 points.

The program then runs the least squares polynomial fitting function (see section **1.2**) on each set ($\mathbf{X_i}$, $\mathbf{Y_i}$)
where

$$0 < i \leq (N \quad `mod` \quad 20)$$

$$N = \textbf{total no. of data points}$$

for each set of the functions specified to fit the data to (see section **1.2**). These results (arrays of coefficients) are stored and then each result is used to calculate a sum squared error. The sum squared error is calculated using the following formula, where $\mathbf{X_i}$ and $\mathbf{Y_i}$ are the sets of 20 data points mentioned previously and $\mathbf{f}$ is a function from the set in section **1.2**:

$$s_\text{i} = \sum_{j=1}^{20} (\mathbf{Y_{ij}} - \mathbf{f_i}(\mathbf{X_{ij}}))^2$$

This set of errors is saved in an array. A weighed array $\mathbf{W}$ is created and the positions of the array correspond to functions in the set in section **1.2**. These weights are multipliers and are applied to the sum squared errors $\mathbf{s}$ to ensure higher-order polynomial functions and non-linear/non-polynomial functions are less likely (higher-order/non-polynomial functions have larger weights in the array $\mathbf{W}$). The weights are multiplied with their respective $\mathbf{s}$ values; the order of this $\mathbf{W}$ array is [linear, quadratic, cubic, quartic, sine]. The minimum value in this new array of weighted sum squared errors is then stored as a new variable $\mathbf{S_s}$. The idea behind these weights and their implementation is further explored in section **3.1**.

A line of best fit is then calculated for each ($\mathbf{X_i}$, $\mathbf{Y_i}$) by creating a set of 20 $\mathbf{x}$ values equally spaced apart and calculating the resulting $\mathbf{y}$ values by plugging the $\mathbf{x}$ values into the best-fitting function's formula. This new set of ($\mathbf{x}$, $\mathbf{y}$) values is plotted and the sum squared error values $\mathbf{s_i}$ are summed to produce a total error value $\mathbf{S}$.

1.2 Least squares regression

The program runs a generalised least squares regression/polynomial fitting function on each set of data-points ($\mathbf{X_i}$, $\mathbf{Y_i}$) for **linear, quadratic, cubic, quartic and sine functions**. After splitting the data ($\mathbf{X}$, $\mathbf{Y}$) into arrays of length 20, the program runs this function on each array to return a maximum likelihood estimate $\mathbf{wh}$ for each duple array ($\mathbf{X_i}$, $\mathbf{Y_i}$). This maximum likelihood estimation is given by the following equation in terms of matrix transformations and operations

$$\mathbf{wh} = \mathbf{X}^T\mathbf{X}^{-1}\mathbf{X}^T\mathbf{y}$$

and it is derived from finding the maximum argument of the log-probability that the output set $\mathbf{Y}$ is derived from the input set $\mathbf{X}$. This probability can be written as a multivariate Gaussian over the outputs $\mathbf{Y}$ where $\mathbf{w}$ is a likelihood estimate:

$$Py|X(\mathbf{w}) = N_\mathbf{y}(\mathbf{X}\mathbf{w}, I\sigma^2)$$

The **argmax** of the log of this probability gives us the maximum likelihood estimate $\mathbf{wh}$. To get the **argmax**, we must differentiate the log-probability with respect to $\mathbf{w}$, and then set the differential equal to 0:

$$\mathbf{wh} = argmax_\mathbf{w} log(Py|X(\mathbf{w})) = \frac{\mathrm{d}}{\mathrm{d}w} log(Py|X(\mathbf{w})) = 0$$

If we put everything in matrix notation and set **wh**=**w**, we get back the equation for **wh**.

My least squares polynomial fitting function is generalised and takes in 4 parameters; a set of data points **X**, a set of data points **Y**, an integer variable **order** and a Boolean variable **pol** which indicates whether the function to test the data against is a polynomial or a sine function (**true**/**false** respectively). It takes in the extra parameters so it can be easily extended to test the data against higher-order polynomial functions or other unknown functions (cosine, etc).

2 Tests and results

I have tested my algorithm on 11 different test files located in the specified "train_data" folder; each with a different degree of complexity, variation and spread. Below is a table with columns showcasing a visualisation of the best-fitting function my algorithm returns, along with the error values of each 20-data-point segment ($\mathbf{S_i}$) and the total error value of the file **S**.

$$\mathbf{S_i}, \quad 0 < i \leq (N \quad `mod` \quad 20)$$

The visualisation consists of - from left -> right, top -> down; the graphs "basic_1-5", "adv_1-3" and then comparisons between **W1** and **W3** on "noise_1-3", and below that are labelled (by filename) tables containing columns for **S**, the $\mathbf{S_i}$ values and $\mathbf{S_{Wk}}$ representing the weighted vector configurations **W1**, **W2** and **W3**). The error values are rounded to 4 decimal places.

basic_X [Error values were identical for $\mathbf{W_1}$, $\mathbf{W_2}$, $\mathbf{W_3}$]:

Filename	S	$\mathbf{S_1}$	$\mathbf{S_2}$
basic_1	1.1833e-28	1.1833e-28	
basic_2	6.2020e-27	1.4523e-27	4.7497e-27
basic_3	1.1833e-28	1.1833e-28	
basic_4	1.3510e-12	2.7493e-29	1.3510e-12
basic_5	9.2088e-26	9.2088e-26	

adv_1

W [W configuration]	S	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
$\mathbf{W_1}$ [1, 1, 1, 1, 1]	193.9070	58.0723	52.8347	83.0000
$\mathbf{W_2}$ [1, 2, 4, 8, 8]	247.3064	95.6317	56.9527	94.7220
$\mathbf{W_3}$ [1, 1.5, 2, 4, 1]	230.8326	79.1579	56.9527	94.7220

adv_2

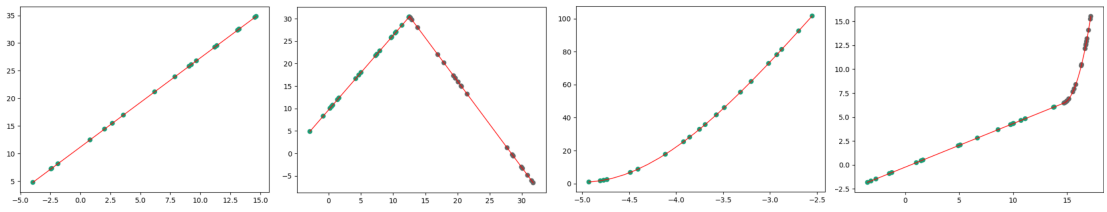
W [W configuration]	S	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
$\mathbf{W_1}$ [1, 1, 1, 1, 1]	3.3432	1.0270	1.3932	0.9230
$\mathbf{W_2}$ [1, 2, 4, 8, 8]	4.1937	1.2142	1.4710	1.5085
$\mathbf{W_3}$ [1, 1.5, 2, 4, 1]	3.6277	1.0270	1.4710	1.1297

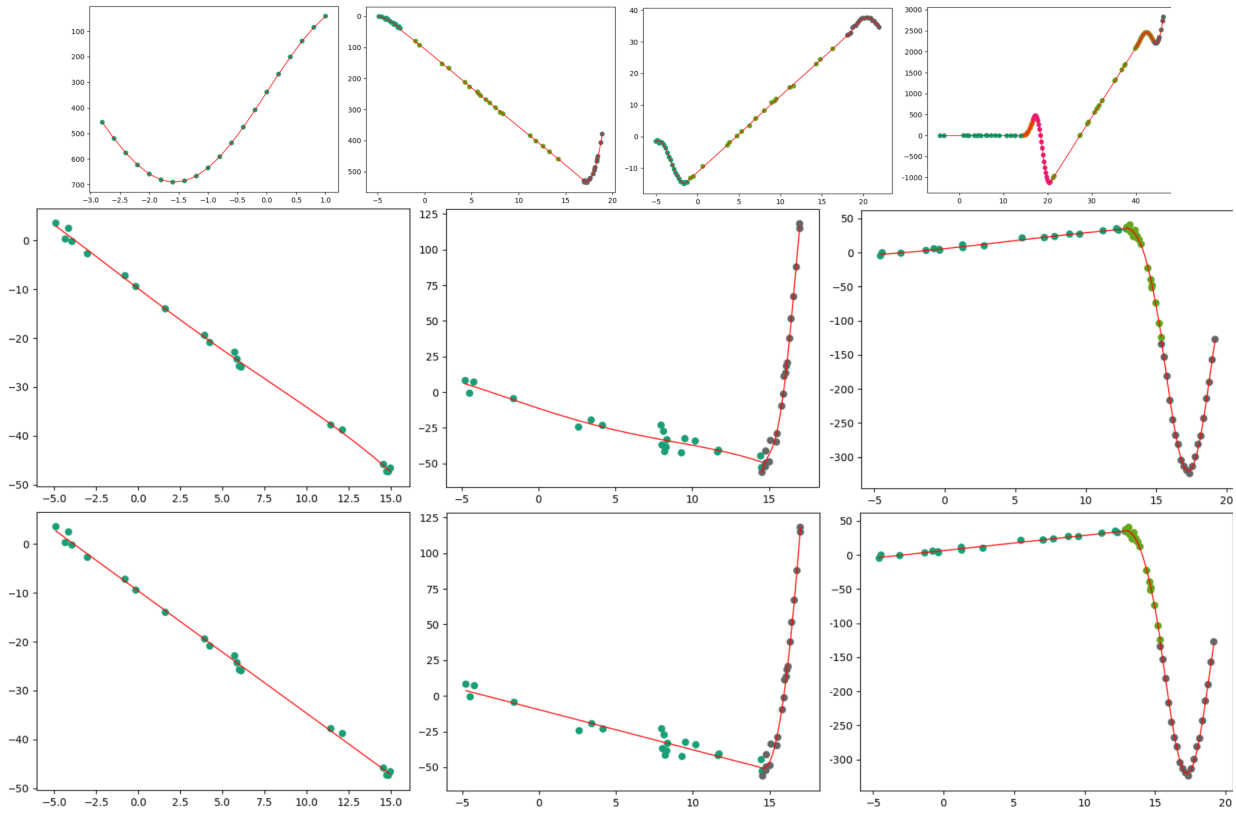
noise_1

W [W configuration]	S	$\mathbf{S_1}$
$\mathbf{W_1}$ [1, 1, 1, 1, 1]	10.0574	10.0574
$\mathbf{W_2}$ [1, 2, 4, 8, 8]	11.9302	11.9302
$\mathbf{W_3}$ [1, 1.5, 2, 4, 1]	11.9302	11.9302

noise_2

W [W configuration]	S	$\mathbf{S_1}$	$\mathbf{S_2}$
$\mathbf{W_1}$ [1, 1, 1, 1, 1]	710.5314	402.1938	308.3377
$\mathbf{W_2}$ [1, 2, 4, 8, 8]	836.1404	466.6386	369.5018
$\mathbf{W_3}$ [1, 1.5, 2, 4, 1]	836.1404	466.6386	369.5018





3 Analysis and improvements

3.1 Overfitting and Generalisation

Depending on which **W** configuration the data is weighted with, the model I have created can overfit. However, the algorithm that classifies data points into a function does prioritise simpler functions. There is an if/else statement which plots a function based on the minimum weighted sum squared error. This means the results are highly dependant on **W**.

The benefit of having a configurable **W** weight vector is that overfitting can be controlled. By giving more complex, unlikely functions (cubics, quartics, etc) higher **w** values in the **W** vector these function's error value is higher and so the algorithm may classify the data differently and provide a different function; one that may fit the actual signal better rather than simply choosing the smallest error value.

Examples of this show up when using the model on "train_data"; specifically the files "noise_1" through "noise_3" (see last 6 graphs above). The vector $\mathbf{W_1}$=[1, 1, 1, 1, 1] was applied to the errors in the top 3 graphs, and $\mathbf{W_3}$=[1, 1.5, 2, 4, 1] was applied in the bottom 3 graphs. "noise_1" is more likely to be a linear relationship rather than a shallow cubic, and so applying $\mathbf{W_3}$ results in a better estimate. This also results in smoother curves as a result, as in "noise_2/3" the transition between line segments is smoother. The model generalises quite well and rarely encounters an overfitting error; the model could also be extended easily to utilise cross-validation and tweak the values of **W** such that it overfits as little as possible; by reducing the chance of overfitting and producing functions which have large orders and small coefficients.

3.2 Extensions

I have included a "–help" parameter to the program which displays the utility functions, the parameters to be supplied to the terminal and their purposes ("–detail", "–plot", "–help"). The "–detail" parameter makes the program print out the error values for each line segment in the signal for detailed analysis, which is how I got the $\mathbf{S_i}$ values earlier.

The least squares polynomial regression and the sum squared error functions are also general and allow for higher orders than quartic (order 4) and are easily extendable for more "unknown"/non-polynomial functions such as cosine, tan, log, ln etc. This could be further extended to allow the model to generate new functions to test the data against, applying weights of at least the size of the next-largest-order function's weight. Limitations on function coefficients and comparisons to previous line segment classifications is also easy to implement to improve the model, and this could be woven in with supervised learning techniques and cross-validation to produce a more machine-learning oriented model which improves over a series of generations (iterations).