1 Introduction

This is a report analysing the methods I used to implement a program that creates models for datasets provided by creating regression lines produced by the Least Squares Method (LSR)

2 Least Squares Method

The 'Least Squares Method' plots a a regression line, assuming a regular function - in a certain form (i.e. linear form would be: $y = a + bx$), as a model for a dataset. In order for an accurate model we define:

$$\mathbf{Y} = \begin{bmatrix} Y_0 \\ \dots \\ Y_n \end{bmatrix} \quad \hat{\mathbf{Y}} = \begin{bmatrix} a_0 + bX_0 + \dots + nX_0^n \\ \dots \\ a_n + bX_n + \dots + nX_n^n \end{bmatrix}$$

And use this absolute distance equation:

$$||\mathbf{Y} - \hat{\mathbf{Y}}||^2 = 0 \quad \textbf{where} \quad \hat{\mathbf{Y}} = \mathbf{X} \cdot \mathbf{A} \tag{1}$$

to minimise the distance between the plot created by the regression line ($\hat{\mathbf{Y}}$) and the actual training dataset($\mathbf{Y}$). Where $\mathbf{X}$ represents the values of x from the dataset given, and $\mathbf{A}$ would represent the coefficients used in the assumed form.

$$\mathbf{X} = \begin{bmatrix} 1_{00} & x_{01} & \dots & \mathrm{x}^n{}_{0p} \\ \dots & \dots & \dots & \\ 1_{n0} & x_{n1} & \dots & \mathrm{x}^n{}_{np} \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} a \\ \dots \\ n \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} 1_{00} & x_{01} & \dots & \mathrm{x}^n{}_{0p} \\ \dots & \dots & \dots & \\ 1_{n0} & x_{n1} & \dots & \mathrm{x}^n{}_{np} \end{bmatrix} \bullet \begin{bmatrix} a \\ \dots \\ n \end{bmatrix}$$

Which will allow us to rewrite the distance equation as:

$$\mathbf{Y} - \mathbf{X} \cdot \mathbf{A} = 0 \Rightarrow \quad \mathbf{X.T} \cdot \mathbf{Y} = \mathbf{X.T} \cdot \mathbf{X} \cdot \mathbf{A} \quad \Rightarrow \quad (\mathbf{X.T} \cdot \mathbf{X})^{-1} \cdot \mathbf{Y} = \mathbf{A} \tag{2}$$

Given that we know $\mathbf{Y}$ and , in order minimise this distance equation, we must compute the values of the coefficients in the assumed form. Solving the distance equation, we end up with equation: $\mathbf{Y} = \mathbf{X} \cdot \mathbf{A}$. However, we cannot guarantee that $\mathbf{X}$ is invertible, and so must multiply both sides by a $\mathbf{X.T}$ (transposed $\mathbf{X}$) to ensure it's a square and invertible matrix. We can then multiply both sides by the inverse and solve for $\mathbf{A}$.

3 Implementation

My implementation uses a very adaptable approach to being able to test many types of functions and return the best ones for each 20 data point segments. The most complex function in 'lsr.py' is my LSR method for polynomials. This method was implemented to be adjustable in the sense that it would (in addition to the dataset's x and y values) take in a 'poly' argument, which determined the power of polynomial it would be plotting.

Listing 1: LSR method for Polynomial

```
...
for p in range(1, poly):
  for i in range(len(xs)):
    xValue = np.power(xs[i], p)
    xArray[i, 0] = xValue
  powColumns = np.concatenate((powColumns,
      xArray), axis=1)
  w = np.linalg.inv(np.dot(
       powColumns.T, powColumns))
  a = np.dot(np.dot(w, powColumns.T), ys)
```

Listing 2: Plotting Regression Using '**a**'

```
  a = np.dot(np.dot(w, powColumns.T), ys)
...
for x in range(len(xs)):
  for i in range(len(a)):
    y_val = y_val + a[i] * pow(xs[x], i)
    y_plot = np.append(y_plot, y_val)
    y_val = 0.

...
return y_plot, a
```

Listing 1 (**L1**) shows the construction of column vectors containing the x values from the dataset being put to a certain power, then concatenated together to create the $\mathbf{X}$ matrix used in LSR, resulting in (depending on the

'poly' value passed into the function) a variable sized vector **a** which represents the coefficients of the polynomial (e.g. poly=3 $\Rightarrow$ **a** = [a, b, c]). Listing 2 (**L2**) shows the implementation of the plotting of a regression line using '**a**' from **L1**. The 'y_plots' are then used to calculate the squared error (SE) of the regression line of that specific polynomial power.
As for the other function forms I tested ('Exponential' and 'Sine'), I essentially treated them as if they were linear equations, this time creating a predetermined $n \times 2$ matrix with a constant column and the respective unknown function of x as the 2nd column. Using similar plotting methods as **L2** to create 'y_plots' for the unknown function.
When it came to testing, I placed all the functions in the testing framework:

Listing 3: LSR method call for Polynomial

```
for l in range(len(sampleX)):
...
  for p in range(1, limit+1):
    y_plot, a = leastSquaresPoly(sampleX[l], sampleY[l], p)
    error, y_set, a = squaredError(y_plot, ys[l*segmentLength: l*segmentLength+segmentLength], a)
```

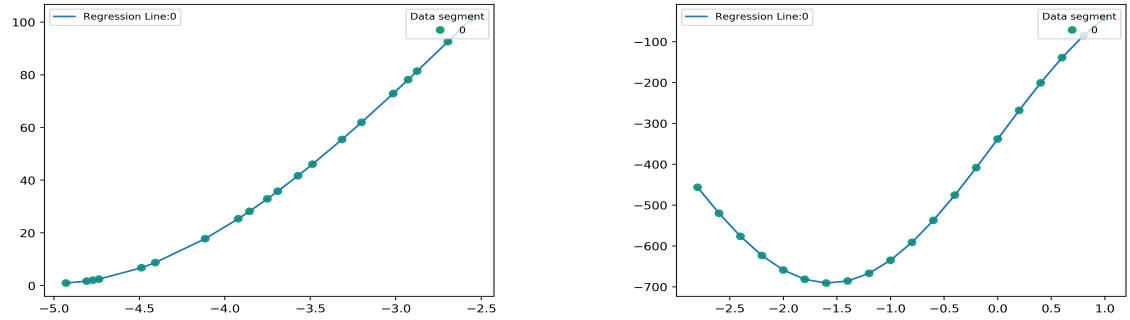
Listing 4: LSR method call for Unknown functions

```
#Sine function
  y_plot, a = leastSquaresSin(sampleX[l], sampleY[l])
  sinePlot = np.array(y_plot)
  error, y_set, a = squaredError(sinePlot, ys[], a)
#Exp function
  y_plot, a = leastSquaresExp(sampleX[l], sampleY[l])
  expPlot = np.array(y_plot)
  error, y_set, a = squaredError(y_plot, ys[segment], a)
```

I would take the 'errors' and 'y_set' (set of $\hat{\mathbf{y}}$ made from the regression lines and place them in lists. The original plan was to select the smallest error from the set of errors given by all functions- then use it to plot the corresponding 'y_set' as the regression for that 20 point segment (with its respective error being the one to contribute to the total error). However, that approach led to drastic overfitting as the larger power polynomials would be afforded the flexibility to better fit their shapes around the data points of **Y**, leading to smaller SE with overly complex regression lines. Instead of selecting smallest errors, I went on to create testing frameworks to attach a regulating system that scaled with larger power/ complexity functions. This would deter my program from selecting overfitted regression lines. Through the implementation of a '**scaledErrorArray**' which consisted of SEs that were scaled by multipliers decided by either: the power of the polynomial, or overall complexity of the function (e.g. trigonometric functions need to be penalised as they are more complex than say, linear functions). The regulating system chosen will be discussed in the next section.

4 Testing

Figure 1: Plots for Basic_3 and Basic_5 respectively



For testing purposes, I set the complexity of the polynomials to only reach the limit of quintics (limit = 5) and only implemented sine and exponential LSR methods. Initially, I set the scaling multiplier to be the highest power of each polynomial (for polynomial functions) and for the other complexities, as $\times$ 2.0. I collated a dataset of a SEs made by each function for every segment in each training dataset, and compared their values to what the program chose to be the best fitting line.
Upon inspection of the line being chosen, in most cases, the program chose the (what I suspected to be) correct line for the dataset. Figure 1 shows us the regression line used to plot for dataset Basic_5. It chose to use a sine line as it contained an SE of $1.050E^{-}25$. Some may argue that this could possibly be an overfitted line, but seeing as the next best SE made by other functions was 2.754 (made by a quintic function). I could safely assume that the graph was actually a perfectly fitted Sine graph, especially as even with the current scaling scheme, the program selected the same. From this, I concluded that the sine graph was definitely a function that my program needed to be able to plot as the 'unknown' function.
The challenge came when testing which polynomial my program should have the capability to plot (seeing as the 'linear' and 'unknown' function has already been decided). Upon further inspection of the SE numbers produced by certain functions, it became apparent that the 'quadratic' and 'cubic' functions were producing very similar results in terms of SEs in the more complex tests like Adv_3 and Noise_3:

Table 1: Adv_3 Polynomial SEs (By Segment)

Adv_3	Segment 1	Segment 2	Segment 3	Segment 4	Segment 5	Segment 6
Quadratic	185.1005	159.7411	319861.4	127.8237	3313.754	136.3078
Cubic	184.8123	159.6849	1504.468	116.0942	1415.177	141.3912
Quartic	178.0843	150.8907	1103.702	105.1333	108539.1	8604301
Quintic	157.0354	691.4879	359.9502	102.7228	2892590	8331805

Table 2: Noise_3 Polynomial SEs (By Segment)

Noise_3	Segment 1	Segment 2	Segment 3
Quadratic	48.01845	197.5501	896.2195
Cubic	43.65814	197.3953	894.4915
Quartic	43.62303	190.9749	206.2706
Quintic	43.54537	181.0787	2422.194

Across most of these tables, the 'quadratic' and 'cubic' SEs are relatively close. This would usually be an argument that cubic graphs are overfitting as they achieve similar SEs to a less complex polynomial in a quadratic. Those highlighted cells are cells in which the program selected between the 2 polynomial functions, usually favouring quadratic due to the smaller scaling factor ($\times 2 < \times 3$). However, there was an outlier case in 'Basic_3' (in **Figure 1**) which decided led me to favouring cubics instead. In this training dataset, the SE for cubics was a staggering 1.438483E-18, whereas for quadratics was 15.74331. The reason this couldn't be classed as overfitting by a cubic graph is that the other functions (higher power ones) all had much larger SE values in comparison, leading me to conclude that Basic_3 was a perfectly fitted cubic graph. That and it providing similar SEs to quadratic graphs led me to selecting it as my chosen polynomial function over quadratics and higher powered (probably overfitted) graphs.

5 Conclusions

In my final version of the 'lsr.py', I have elected to include only: linear, cubic and sine graphs as forms of regression lines. This is due to the evidence stated above where the 2 specific test data: Basic_3 and Basic_5. These 2 sets of training data led me to believe that cubic and sine graphs (respectively) were perfect fittings for them, and so had to be the ones to be included. I determined them not to be overfitting graphs because they had massively lower SE values in comparison to the other implemented functions (i.e. more than 10^{-6} smaller than the next best) even with more complex polynomials were tested (which would've definitely overfitted the data). This gap in error value and how close their errors were with simpler functions (in the case of cubics vs quadratics) concluded that they were the appropriate functions to include. The use of my regulating system (scaling the SE values) wasn't the best way of detering overfitting as the scaling multipliers were not all regular (i.e. sine and exp functions were scaled to a predetermined value), however, they did scale errors in a way that favoured simpler function calls and did (for most of the training data) cause the program to call (what I looked to be) the matching function (one with the smallest error but also did not overfit the training data).