An Unknown Signal Report

George Herbert
cj19328@bristol.ac.uk

April 24, 2021

1 Equations for linear regression

For a set of (x, y) coordinates that lie along a line with Gaussian noise, with the relationship $\mathbf{y} = \mathbf{X}\mathbf{w} + \epsilon$ where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, the maximum likelihood estimation of $\mathbf{w}$ is equivalent to the least square error estimation and is given by the equation:

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

This equation is implemented in my code as the following method:

```
def regressionNormalEquation(self, X, y):
    return np.linalg.inv(X.T @ X) @ X.T @ y
```

$\mathbf{X}$ can take one of the following three forms:

$$\mathbf{X} = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_{20} & 1 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x_1^n & x_1^{n-1} & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ x_{20}^n & x_{20}^{n-1} & \dots & 1 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} f(x_1) & 1 \\ \vdots & \vdots \\ f(x_{20}) & 1 \end{bmatrix}$$

depending on whether the line is linear, polynomial of degree n, or the unknown function f, respectively.

2 Choice of polynomial degree

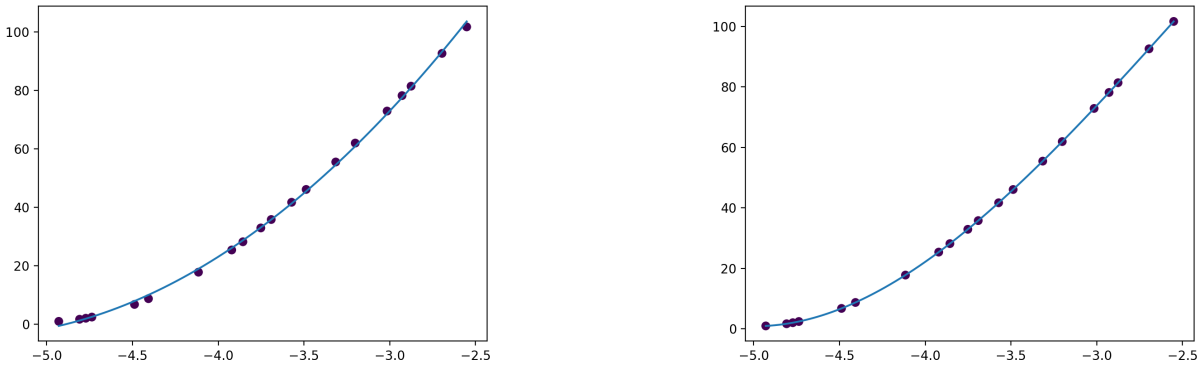
I used several different techniques to determine and validate that the polynomial degree used to genereate the unknown signals was likely cubic.

Firsly, a polynomial of degree n can have a maximum of $n-1$ relative extrema. Across all of the unknown signals, no line segment appeared to have more than two relative extrema. This made it is likely, but not definitive, that the polynomial degree was relatively low (e.g. cubic, quartic). However, there was still a possibility the polynomials used to generate the unknown signals were of a higher degree, and that the segments chosen just happened to display the characteristics of a lower-degree polynomial (e.g. low number of relative extrema).

To better identify exactly which polynomial degree was used to generate the unknown signals, I created a program '`display.py`' to visualise the points. By inspecting the appearance of each segement, I drew up a list of nonlinear segments. I then created a program, '`degree.py`', that calculated the cross-validation error for these line segments when trained using a model with a polynomial of degree 2 to a polynomial of degree 10; Table 1 shows a small section from the output of this program. Having analysed the output, it was clear that a large proportion of the nonlinear signals had their minimum cross-validation error when fitted with a polynomial

Table 1: Section of the output from '`degree.py`'

Filename	Line segment	Polynomial degree	Cross-validation error
basic_3.csv	0	2	7.3947610358752875
basic_3.csv	0	3	1.2989585613760917e-23
⋮	⋮	⋮	⋮
adv_3.csv	5	9	318.8443359827487
adv_3.csv	5	10	279.2750683133305



(a) Quadratic fit

(b) Cubic fit

Figure 1: Two fits for '`basic_3.csv`'

of degree 3. This consistent minimum cross-validation error indicated that the polynomial line segments in the unknown signal are cubic.

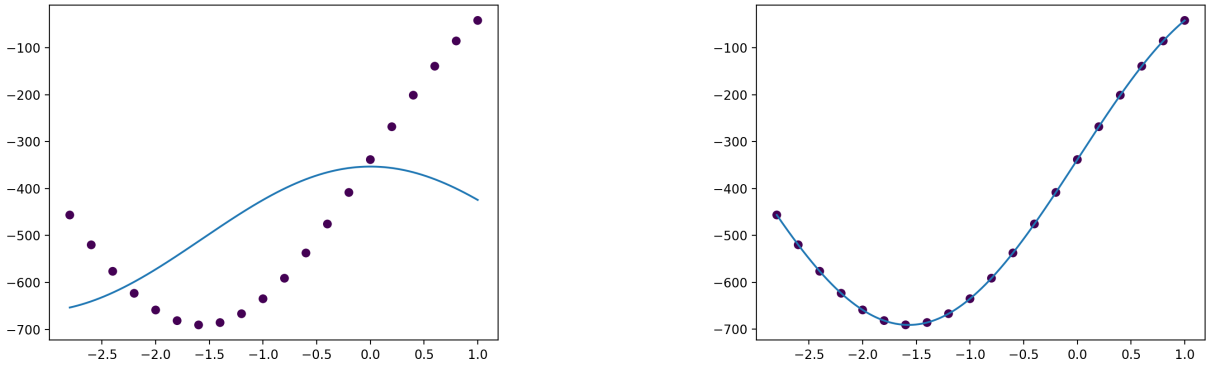
To confirm this visually, I used the '`basic_*.csv`' files as they have a negligable amount of noise. I produced fits using several different polynomials. An example of this is shown in Figure 1. The quadratic fit shown in Figure 1a is clearly not as good a fit as the cubic fit shown in Figure 1b.

3 Choice of unknown function

Using my '`display.py`' program to visualise the signals, I produced a list of potential 'unknown functions' that could represent the underlying signal of line segments based on their shapes: $\mathbf{w}_1 sin(x) + \mathbf{w}_2$, $\mathbf{w}_1 cos(x) + \mathbf{w}_2$, $\mathbf{w}_1 tan(x) + \mathbf{w}_2$ and $\mathbf{w}_1 e^x + \mathbf{w}_2$. I then created a program, '`unknown.py`', that calculated the cross-validation error for each of the nonlinear line segments previously identified when trained using each of the potential unknown functions. A table displayed the cross-validation errors—similar to that used to determine the polynomial degree. Having analysed the cross-validation errors, it was clear that all nonlinear signals that were likely not a cubic polynomial had their minimum cross-validation error when trained to fit the function $\mathbf{w}_1 sin(x) + \mathbf{w}_2$.

However, when trained to fit the function $\mathbf{w}_1 cos(x) + \mathbf{w}_2$ the cross-validation errors were also quite small. To

Having incorporated regression to fit a function of the form $\mathbf{w}_1 sin(x) + \mathbf{w}_2$ into my '`lsr.py`' least-squares regression program, I ran the program on the '`basic_5.csv`' and '`adv_3.csv`' unknown signals; the outputs are shown in Figure 2 and Figure **??** respectively. The lines being



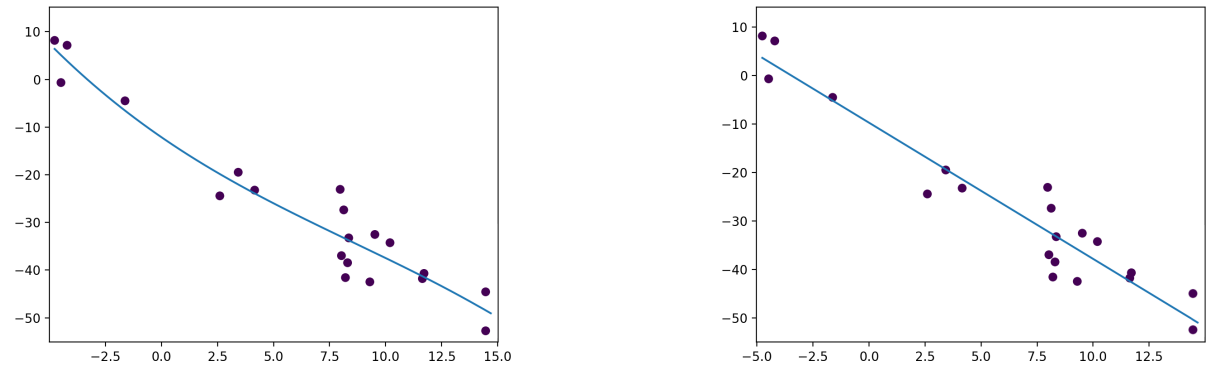
(a) Cosine fit

(b) Sine fit

Figure 2: Two fits for '`basic_3.csv`'

a near-perfect fit allowed me to validate that the unknown function is of the form $\mathbf{w}_1 sin(x) + \mathbf{w}_2$ visually.

4 Model selection



(a) Cubic fit

(b) Linear fit

Figure 3: Two fits for the first line segment in '`noise_2.csv`'

Overfitting occurs when a machine learning algorithm produces a model that has learnt the noise in the data as if it represents the structure of the underlying model [1]. In the case of linear regression, overfitting is most likely to occur by producing a model with too complex a function type, such that it would fail to predict future observations. Figure 3 shows an example of this. On the left, (a) is a cubic fit that is too complex a function type for the data points. It would not predict future y-values for a set of given x-values as successfully as the linear fit (b) on the right.

To prevent overfitting, I have used leave-one-out cross-validation when producing a model for each 20-point line segment. Leave-one-out cross-validation is an extreme case of k-fold cross validation such that $k = n$, where n is the number of data points (in this case, 20). Despite being computationally expensive, I believe that leave-one-out cross-validation is an appropriate

technique to prevent overfitting in this case, owing to the limited sample size of each line segment.

Leave-one-out cross-validation involves using each of the 20 data points exactly once as validation data for a model trained using the other 19 data points. The cross-validation error for each function type is calculated as follows [2]:

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}^{(-i)})^2$$

where n is the number of data points in a line segment (i.e. 20), y_i is the actual y-value for the i-th data point, and $\hat{y}^{(-i)}$ is the predicted y-value for the i-th data point when trained without using the i-th sample.

The function type with the lowest cross-validation error is then selected for each line segment, and the weights $\hat{\mathbf{w}}$ are determined by training on all data points for that segment.

5 Optimisations and improvements

To begin with, computing the matrix inverse using the `np.linalg.inv` method is computationally expensive and unnecessary. Instead, given $\mathbf{X}$ and $\mathbf{y}$, the maximum likelihood estimation of $\mathbf{w}$ could be directly computed as follows: `np.linalg.solve(X.T @ X, X.T @ y)`. Computing $\hat{\mathbf{w}}$ directly would be faster, as `np.linalg.inv` computes the inverse of a matrix $\mathbf{A}$ by solving for $\mathbf{A}^{-1}$ in $\mathbf{A}\mathbf{A}^{-\mathbf{1}} = \mathbf{I}$ [3]. Thus, there would be a performance benefit by solving for $\hat{\mathbf{w}}$ in $\mathbf{X}^T\mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^T\mathbf{y}$ directly.

Another computationally expensive operation in my algorithm is that used to calculate the cross-validation error using leave-one-out cross-validation. The method currently involves fitting the model and calculating the sum squared error n times. Instead, there exists a faster method I could have adopted that involves calculating the leverage. Despite this, I opted not to include this method because my program, as it currently stands, can be easily adapted to use k-fold cross-validation for any value of k that is a factor of 20—changing the constant '`K`' in the code achieves this.

6 Testing

I created a file, '`test.py`', that uses the `unittest` framework to test each of the methods in '`lsr.py`'.

References

[1] Burnham, K. P. and Anderson, D. R. (2002) *Model Selection and Multimodel Inference*. 2nd ed. Springer-Verlag.

[2] Taylor, J. (2020) *Leave one out cross-validation (LOOCV) — STATS 202* https://web.stanford.edu/class/stats202/notes/Resampling/LOOCV.html

[3] Muldal, A. (2017) *Why does numpy.linalg.solve() offer more precise matrix inversions than numpy.linalg.inv()?* https://stackoverflow.com/a/31257909/8540479