Linear Regression and Least-Squares Filtering

Minh N. Do

Consider the linear regression model that predict an output variable y as a linear combination of input variables $x_1, x_2, \ldots, x_K$ as

$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + w_1 x_1 + \ldots + w_K x_K = \underbrace{\begin{bmatrix} 1 & x_1 & x_2 & \ldots & x_K \end{bmatrix}}_{\boldsymbol{x}^T} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_K \end{bmatrix}}_{\boldsymbol{w}}, \tag{1}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_K]^T$ contains the model parameters, including the bias w_0; and $\boldsymbol{x} = [1, x_1, \ldots, x_K]^T$ contains the input variables, including the "dummy" variable $x_0 = 1$ to account for the bias.

Given a training data set comprising of N examples of input $\{\boldsymbol{x}_n\}_{n=0}^{N-1}$ together with corresponding target values $\{t_n\}_{n=0}^{N-1}$, we want to find the best fitted linear regression $y(\boldsymbol{x}, \boldsymbol{w})$ as in (1) to minimize the errors

$$e_n = y(\boldsymbol{x}_n, \boldsymbol{w}) - t_n = \boldsymbol{x}_n^T \boldsymbol{w} - t_n, \quad \text{for } n = 0, 1, \ldots, N-1,$$

in the least-squares sense. That means we need to find the model parameters $\boldsymbol{w}$ to minimize the following objective function

$$J(\boldsymbol{w}) = \sum_{n=0}^{N-1} e_n^2 = \sum_{n=0}^{N-1} (\boldsymbol{x}_n^T \boldsymbol{w} - t_n)^2. \tag{2}$$

We can stack the examples in the training data set and express the error and objective function in the matrix form as

$$\underbrace{\begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \\ e_{N-1} \end{bmatrix}}_{\boldsymbol{e}} = \underbrace{\begin{bmatrix} \boldsymbol{x}_0^T \\ \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_{N-1}^T \end{bmatrix}}_{\boldsymbol{X}^T} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_M \end{bmatrix}}_{\boldsymbol{w}} - \underbrace{\begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_{N-1} \end{bmatrix}}_{\boldsymbol{t}}, \tag{3}$$

and

$$J(\boldsymbol{w}) = \|\boldsymbol{e}\|^2 = \|\boldsymbol{X}^T \boldsymbol{w} - \boldsymbol{t}\|^2. \tag{4}$$

Note that the matrix $\boldsymbol{X}^T$ is of size $N \times (K+1)$. Typically, the number of training samples N is much bigger than the number of parameters $(K+1)$ in $\boldsymbol{w}$ that we need to recover. Once the parameters $\boldsymbol{w}$ are recovered, given a new sample input in $\boldsymbol{x}$, we can estimate the corresponding output y using (1).

In another application setting, suppose that we want to filter an input signal $\{x_n\}$ using an FIR filter $\{w_k\}_{k=0}^K$ to produce an output that matches a desired signal $\{t_n\}$. Let $\{y_n\}$ be the output of the filter, we have

$$\begin{aligned} y_n &= \sum_{k=0}^{K} w_k x_{n-k} \\ &= \underbrace{\begin{bmatrix} x_n & x_{n-1} & x_{n-2} & \dots & x_{n-K} \end{bmatrix}}_{\boldsymbol{x}_n^T} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_K \end{bmatrix}}_{\boldsymbol{w}}. \end{aligned} \tag{5}$$

We want to find the filter coefficients $\{w_k\}_{k=0}^K$ to minimize the error signal

$$e_n = y_n - t_n = \boldsymbol{x}_n^T \boldsymbol{w} - t_n$$

in the least-squares sense as in the linear regression problem above. Similarly, we can stack the samples in matrix form as

$$\underbrace{\begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ \vdots \\ e_{N-1} \end{bmatrix}}_{\boldsymbol{e}} = \underbrace{\begin{bmatrix} \boldsymbol{x}_0^T \\ \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_{N-1}^T \end{bmatrix}}_{\boldsymbol{X}^T} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_K \end{bmatrix}}_{\boldsymbol{w}} - \underbrace{\begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ \vdots \\ t_{N-1} \end{bmatrix}}_{\boldsymbol{t}}. \tag{6}$$

Notice that with $\boldsymbol{x}_n^T$ and $\boldsymbol{X}^T$ defined as in (5) and (6), $\boldsymbol{X}^T$ is the convolution matrix with $\{x[n]\}$.

To minimize the objective function $J(\boldsymbol{w})$ defined in (2), we can take the gradient and set it to zero. Using the chain rule and noticing that the gradient of $f(\boldsymbol{w}) = \boldsymbol{x}^T\boldsymbol{w}$ is $\nabla f(\boldsymbol{w}) = \boldsymbol{x}$, we have

$$\begin{aligned} \nabla J(\boldsymbol{w}) &= \sum_{n=0}^{N-1} 2(\boldsymbol{x}_n^T \boldsymbol{w} - t_n)\boldsymbol{x}_n \\ &= 2\sum_{n=0}^{N-1} e_n \boldsymbol{x}_n \\ &= 2\boldsymbol{X}\boldsymbol{e} \\ &= 2\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{w} - \boldsymbol{t}). \end{aligned} \tag{7}$$

Here $\boldsymbol{X}$ is the transpose of $\boldsymbol{X}^T$ defined as in (3), or

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \dots & \boldsymbol{x}_{N-1} \end{bmatrix}.$$

Therefore the least-squares filter $\boldsymbol{w}$ satisfies the following equation

$$\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{w} = \boldsymbol{X}\boldsymbol{t}. \tag{8}$$

Thus,

$$\boldsymbol{w} = \left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1}\boldsymbol{X}\boldsymbol{t}. \tag{9}$$

While (9) provides a closed-form formula for the desired least-squares linear regression or filtering, sometimes it is not practical as it requires the delay, storage, and multiplication and inverse with a big data matrix. To circumvent these issues, we can resort to the gradient-descent method that iteratively refine an estimate of $\boldsymbol{w}$ in the steepest (i.e. negative gradient) direction:

$$\boldsymbol{w}_{l+1} = \boldsymbol{w}_l - \frac{\mu}{2}\nabla J(\boldsymbol{w}_l), \quad \text{for } l = 0, 1, 2, \ldots, \tag{10}$$

where μ ($\mu > 0$) is called the step size or learning rate.

Substituting the gradient from (7) to (10), we obtain the least-squares update:

$$\begin{aligned}\boldsymbol{w}_{l+1} &= \boldsymbol{w}_l - \mu\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{w}_l - \boldsymbol{t})\\ &= \boldsymbol{w}_l - \mu\sum_{n=0}^{N-1}(\boldsymbol{x}_n^T\boldsymbol{w}_l - t_n)\boldsymbol{x}_n.\end{aligned} \tag{11}$$

Sometimes computing the gradient in (11) is still too expensive or requires long delay for the sum over N data samples. A further approximation is to replace the gradient sum over all data samples to only over a small subset of samples, in particular using only one sample:

$$\boldsymbol{w}_{l+1} = \boldsymbol{w}_l - \mu(\boldsymbol{x}_l^T\boldsymbol{w}_l - t_l)\boldsymbol{x}_l. \tag{12}$$

This approximation is called stochastic gradient descent (SGD) method and the resulting update (12) is the famous least mean squares (LMS) algorithm. SGD is the main training algorithm for many current machine learning methods including deep learning. The key advantage of LMS is that it can be used on-line and used adaptively. Each LMS iteration takes a new data sample $\boldsymbol{x}_l$ and produces a prediction based on the current model parameter $\boldsymbol{w}_l$ as

$$\widehat{t_l} = y_l = \boldsymbol{x}_l^T\boldsymbol{w}_l.$$

Then based on the prediction error, the model parameter is updated according to (12) for the next iteration and data sample. There is no need to wait for all training data to become available. As a result, even if the underlying model parameter changes, the LMS update (12) can automatically track this change and adapt the estimated model parameters accordingly.

The following Python script demonstrates the LMS algorithm.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
%matplotlib inline

N = 400                                # Input size
K = 31                                 # Filter size
```

```
x = np.random.randn(N)                 # Input to the filter
h = signal.firwin(K, 0.5)              # FIR system to be identified
t = signal.convolve(x, h)              # Target output signal
t = t + 0.01 * np.random.randn(len(t)) # with added noise

mu = 0.05                              # LMS step size

fig = plt.figure()
plt.title('Unknown filter')
plt.stem(h, 'r')

w = np.zeros(K)                        # Initial filter
e = np.zeros(N-K)

for n in range(0, N-K):
    xn = x[n+K:n:-1]
    en = t[n+K] - np.dot(xn , w)       # Error
    w = w + mu * en * xn               # Update filter (LMS algorithm)

    e[n] = en                          # Record error

    # Plot updated filter after each iteration
    if (n % 50 == 0):
        plt.figure()
        plt.title('Estimated filter at iteration %d' % n)
        plt.stem(w, 'b')

plt.figure()
plt.title('Error signal')
plt.stem(e, 'b')
```

Exercises

1. Run the above script and compare the estimated filter `w` with ground truth filter `h`. Try with different filters `h`, including with different size and coefficients.

2. Run the script and comment on the convergence of the LMS algorithm with different values of step size `mu`, including a negative value.

References

[1] C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006.

[2] T. K. Moon and W. C. Stirling, *Mathematical Methods and Algorithms for Signal Processing*, Pearson, 1999.