Fitting points to lines

Alex S.[*]

1 Least Squares

The gist of the process is: we define a model and a loss function that measures how well our model fits the data. We then minimize (by solving a system of linear equations) the loss function over model parameters, thus finding a model (plane) that fits the data well.

Let us define our model $f_{w_1,\ldots,w_N}$ as:

$$f(x) = \sum_{i=1}^{N} w_i x_i$$

In other words, our model is a hyperplane in N-dimensions. If we assume exact training data S, we have:

$$\boldsymbol{Xw} = \boldsymbol{y}$$

We can solve this for $\boldsymbol{w}$ by solving a system of linear equations. Unfortunately real world data generally has sampling errors, and wouldn't perfectly fit a hyperplane (and we wouldn't be able to solve $\boldsymbol{Xw} = \boldsymbol{y}$ for $\boldsymbol{w}$). We can represent the error with ξ:

$$\boldsymbol{\xi} = \boldsymbol{y} - \boldsymbol{Xw}$$

If this error is normally distributed[1], then we can use the *least squares* method to solve for $\boldsymbol{w}$. We start by defining the least squares loss function[2]:

$$\mathcal{L}(w_1,\ldots,w_M) = \sum_{i=1}^{L} (y_i - f_{w_1,\ldots,w_M}(\boldsymbol{x}_i))^2$$

[*] `alex@theparticle.com`

[1] Often, when error appears to be normally distributed, it is really log-normally distributed.

[2] The least squares loss function is derived from an assumption that the sample data set S is generated by a smooth function with Gaussian noise. The probability of the sample data S, is a product of probabilities of individual points, which is proportional to

$$P \propto \prod_{i=1}^{L} e^{-\frac{1}{2}\left(\frac{y_i - f_{w_1,\ldots,w_M}(\boldsymbol{x}_i)}{\sigma}\right)^2}$$

where σ is the sample data standard deviation, which we assume to be constant. Maximizing this probability is equivalent to minimizing the negative of its logarithm, which is equivalent to the sum of squares loss function.

This is essentially ξ^2, i.e.:

$$(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

To find where $\mathcal{L}$ is lowest, we take its derivative with respect to $\boldsymbol{w}$ and set that derivative to 0. We end up with N equations of the form:

$$0 = \sum_{i=0}^{L} \left[y_i - \sum_{j=1}^{N} w_j \boldsymbol{x}_{ij} \right] \boldsymbol{x}_{ik}$$

for $k = 1, \ldots, N$. Rewriting this leads to what are called *normal equations*:

$$\sum_{j=1}^{N} \alpha_{kj} w_j = \beta_k$$

where:

$$\alpha_{kj} = \sum_{i=1}^{N} \boldsymbol{x}_{ij}\boldsymbol{x}_{ik} \qquad \text{and} \qquad \beta_k = \sum_{i=1}^{N} y_i \boldsymbol{x}_{ik}$$

In matrix notation, the above is:

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \qquad \text{which leads to:} \qquad \boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

We often add $\lambda\boldsymbol{I}$ to $\boldsymbol{X}^T\boldsymbol{X}$ to ensure the inverse exists:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This is what is known as a *primal* form. Every linear problem with N variables and M constraints has a corresponding *dual* problem with M variables and N constraints. One is in some sense a transpose of the other. It is sometimes computationally convenient to solve the *dual* problem instead of the primal.

The dual solution first computes what is known as a *Gram matrix*. For a set of vectors $v_1, \ldots, v_n$, the Gram (or Gramian) matrix is a symmetric matrix of inner products, where each entry $G_{ij} = v_i \cdot v_j$. In matrix form, this is simply $\boldsymbol{X}\boldsymbol{X}^T$. We then define:

$$\boldsymbol{\alpha} = (\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Where $\boldsymbol{\alpha}$ are Lagrange multipliers. To get $\boldsymbol{w}$ as in the primal solution we use:

$$\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{\alpha}$$

The major difference between these methods is that the primal operates on $\boldsymbol{X}^T\boldsymbol{X}$ while dual operates on $\boldsymbol{X}\boldsymbol{X}^T$. The size of $\boldsymbol{X}^T\boldsymbol{X}$ grows with dimensions, while the size of $\boldsymbol{X}\boldsymbol{X}^T$ grows with the number of inputs. If $\boldsymbol{X}$ is N by M matrix, then for $N > M$, it becomes more efficient to do primal method, and for $N < M$ to do the dual method.

2 Example

Lets do an example: find a line that passes through points $(2, 13)$ and $(3, 17)$. We need to solve for a and b in the below equations:

$$2a + b = 13$$

$$3a + b = 17$$

Such things are easier to write in matrix form:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} 13 \\ 17 \end{bmatrix}$$

What we have is a classic equation: $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$, where we need to solve for $\boldsymbol{w}$. Rearranging things a bit, we end up with two solutions for $\boldsymbol{w}$

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

$$\boldsymbol{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{y}$$

With these, we learn that $\boldsymbol{w}$ is:

$$\begin{bmatrix} 5 \\ 4 \end{bmatrix}$$

In other words, the line is: $4x + 5 = y$. Octave code for above:

```
X = [1, 2; 1, 3]
y = [ 13 ; 17 ]
w = inv(X'*X + eye(2)*0.001)*X'*y
```

We can use this method to solve any such linear system! Let us do a slightly more complicated example: find a line that passes through points $(2, 13)$, $(3, 17)$, $(5, 23)$, $(7, 29)$, $(11, 31)$, $(13, 37)$. We need to solve for a and b in the below equations:

$$2a + b = 13$$

$$3a + b = 17$$

$$5a + b = 23$$

$$7a + b = 29$$

$$11a + b = 31$$

$$13a + b = 37$$

Rewriting in matrix form:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} 13 \\ 17 \\ 23 \\ 29 \\ 31 \\ 37 \end{bmatrix}$$

Plugging in solutions for $\boldsymbol{w}$, we get:

$$\begin{bmatrix} 11.4437 \\ 1.9836 \end{bmatrix}$$

In other words, the line is approximately: $1.9836x + 11.4437 = y$. Notice that this line does not fit any of the points perfectly, yet it approximately fits all of them! Octave code for above:

```
X = [1, 2; 1, 3; 1, 5; 1, 7; 1, 11; 1, 13 ]
y = [ 13 ; 17 ; 23; 29; 31; 37 ]
w = inv(X'*X + eye(2)*0.001)*X'*y
```

So why the two solutions? The $\boldsymbol{X}^T\boldsymbol{X}$, or primal solution, needs to invert a 2×2 matrix, while the $\boldsymbol{X}\boldsymbol{X}^T$, or dual solution, needs to invert a 6×6 matrix—in this case, doing the primal solution is much faster. For situations when we have few points in many dimensions, solving the dual solution is often faster. For example:

$$\begin{bmatrix} 1 & 2 & 3 & 5 & 7 \\ 1 & 3 & 5 & 7 & 11 \\ 1 & 5 & 7 & 11 & 13 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

Solving for $\boldsymbol{w}$ is much simpler via the dual method ($\boldsymbol{X}\boldsymbol{X}^T$) as it only requires inverting a 3×3 matrix, and not the 5×5 matrix required by the primal ($\boldsymbol{X}^T\boldsymbol{X}$) solution. The solution by either method is

$$\begin{bmatrix} 1.627295 \\ -0.075596 \\ -1.320899 \\ 1.476102 \\ -0.556916 \end{bmatrix}$$

Notice that the above isn't a line; but a hyperplane! Octave code for dual solution above:

```
X = [1, 2, 3, 5, 7; 1, 3, 5, 7, 11; 1, 5, 7, 11, 13]
y = [ 1; -1; 1 ]
w = X'*inv(X*X' + eye(3)*0.001)*y
```

3 Fitting Polynomials

Without any changes, the above method can be used to fit n-degree polynomials. The input matrix $\boldsymbol{X}$ is setup with rows like:

$$\left[x^0, x^1, \ldots, x^n\right]$$

For example, using just x^0 fits a 0th degree polynomial, which is essentially $y = C$, where C is some constant. The 1st degree polynomial is a straight line, $y = Ax + C$, 2nd degree polynomial is $y = Ax^2 + Bx + C$, which is a parabola, etc.

For example, instead of X as

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix}$$

which would fit a line, you can fit a 3rd degree polynomial just by tweaking that matrix to be:

$$\begin{bmatrix} 1 & 2 & 2^2 & 2^3 \\ 1 & 3 & 3^2 & 3^3 \\ 1 & 5 & 5^2 & 5^3 \\ 1 & 7 & 7^2 & 7^3 \\ 1 & 11 & 11^2 & 11^3 \\ 1 & 13 & 13^2 & 13^3 \end{bmatrix}$$

The resulting solutions will have the form $y = D + Cx + Bx^2 + Ax^3$. This can be extended to any degree polynomial you care to fit.

4 Fitting non-linear functions

Now for a bit of magic: this linear method can fit non-linear functions, via the process of embedding. For example, if you want to fit $y = Be^{Ax}$, we can take log of both sides to get $\ln(y) = Ax + \ln(B)$, which is linear (notice that it has the form of $y = Ax + B$). Now you simply use that form in $\boldsymbol{X}$ and $\boldsymbol{y}$ and what you're fitting will be the non-linear $y = Be^{Ax}$.

Similarly, to fit power function $y = B * x^a$ you can take log of both sides to get $\ln(y) = \ln(B) + a * \ln(x)$, which is now also linear.

This embedding is very powerful; the general approach is "kernel methods". The idea is to embed your non-linear data into some higher dimensional space that perhaps has linear structures, and then use a linear solver.