Linear Regression

Sia

July 2025

1 Disclaimer

This document exists only to clarify my own thoughts.

2 Introduction

The point of linear regression is to predict something, e.g. how much money we expect someone to make, given a certain height. It is a linear approximation of the world, and many things aren't that linear, but the trope still holds: some flawed models are useful.

I used matrix and vector notation in this document. Initially, I felt this was better because I thought it mirrors how we code efficient implementations in NumPy. I also liked that it was succinct. Now, however, I think that the notation reveals some things but also obfuscates other things. I guess there are no free lunches - even when it comes to notation. Anyway, I stuck with the notation but realised that it made writing this document harder. Most of our high school maths, and even the odd calculus class we take at university, focus on scalar examples. So while many of us have memorised what the derivative of x^{-1} is, few know what the derivative of $\mathbf{X}^T$ is. Surprisingly, there are few books that teach derivatives from the vector and matrix perspective, though there are cheatsheets.

3 Forward pass

To make predictions, we take inputs and calculate a dot product with weights.

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} \tag{1}$$

where $\mathbf{X}$ is a $N \times D$ input matrix with N examples and D features, and $\mathbf{w}$ is a column vector with shape $D \times 1$. The predictions are $\hat{\mathbf{y}}$.

The linear regression model needs a bias term, or else only a subset of linear equations will be explored, which likely will result in suboptimal predictions. When coding the bias term, a neat trick is to add a column of $1s$ as the first column of the input matrix X, which then becomes an $N \times (D+1)$ matrix, and one additional weight to the weight vector, which is then really a $(D+1) \times 1$ column vector.

4 Backward pass

The weights are in practice randomly initiated or initiated as 0s, which means the first set of predictions are likely bad. We want to adjust the weights to make predictions better - "better" meaning they are closer to the true targets in the dataset. The regression model, therefore, needs to learn, and for that it needs a metric to show how well its predictions did. For this, I used the mean squared error (MSE):

$$MSE = \frac{1}{2N} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 \tag{2}$$

The vertical bar notation is called the squared Euclidean norm and means we are taking every index of $\hat{y}$, subtract the corresponding index of y, square the difference, and then sum all the terms.

We then need to find a method to reduce the error. The solution to this is the shape of this error: if it were plotted on a graph, it would be a parabola. This means that somewhere there is a minimum error, given possible values for the components of the weight vector, and we can find these possible values using derivatives.

5 Finding the best weights in one go

Honestly, its not quite clear to me what the derivative really is. There are different views, and calculus classes tend to focus on mechanical computation, rather than deep understanding of concepts. I guess for ML purposes it can be thought of as an indicator of what the weights need to be, or where they need to be adjusted to, so that you minimise error. One way of using derivatives to figure out what the weights should be is to find the derivative, or more correctly the partial derivatives, of the MSE function and set them equal to 0. The partial derivatives of the MSE itself, also called the gradient, are:

$$\frac{\partial}{\partial \mathbf{w}} MSE = \frac{\partial}{\partial \mathbf{w}} \frac{1}{2N} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 \tag{3}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \tag{4}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) \tag{5}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} (\mathbf{w}^T \mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\mathbf{w} - \mathbf{y}) \tag{6}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{w}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\mathbf{w} + \mathbf{y}^T \mathbf{y} \tag{7}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} tr(\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{w}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\mathbf{w} + \mathbf{y}^T \mathbf{y}) \tag{8}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \Big(tr(\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w}) - tr(\mathbf{w}^T \mathbf{X}^T \mathbf{y}) - tr(\mathbf{y}^T \mathbf{X}\mathbf{w}) + tr(\mathbf{y}^T \mathbf{y}) \Big) \tag{9}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \Big(tr(\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w}) - tr(\mathbf{y}^T \mathbf{X}\mathbf{w}) - tr(\mathbf{y}^T \mathbf{X}\mathbf{w}) + tr(\mathbf{y}^T \mathbf{y}) \Big) \tag{10}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \Big(tr(\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w}) - 2tr(\mathbf{y}^T \mathbf{X}\mathbf{w}) + tr(\mathbf{y}^T \mathbf{y}) \Big) \tag{11}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} \Big(tr(\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w}) - 2tr(\mathbf{w}^T \mathbf{X}^T \mathbf{y}) + tr(\mathbf{y}^T \mathbf{y}) \Big) \tag{12}$$

$$= \frac{1}{2N} \frac{\partial}{\partial \mathbf{w}} (\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \mathbf{y}^T \mathbf{y}) \tag{13}$$

$$= \frac{1}{2N} (2\mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{X}^T \mathbf{y}) \tag{14}$$

$$= \frac{1}{N} (\mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{X}^T \mathbf{y}) \tag{15}$$

Here is what's happening in brief, skipping the more obvious points. The Euclidean norm of $\mathbf{X}\mathbf{w} - \mathbf{y}$ is equal to the dot product of the inner term with its transpose (5). After we expand the product of the terms (7), the entire equation will result in a scalar. We therefore use a trick, and apply the trace t equation (8). A trace is the sum of the diagonals of a matrix, and we can apply the trace because a scalar is a 1×1 matrix and therefore square, and the trace of a scalar is the scalar itself. Really, using the trace is a counterintuitive trick that one just has to have seen before, which makes the final equation more functional, i.e. the partial derivatives are arranged in a neat column vector.

Each individual term in the equation also happens to be a square matrix, and therefore we can use properties of the trace to clean up the equation. For example, the trace of a square matrix is equal to the trace of its transpose (10), which means the two middle terms can be combined (11). Finally, we look for the derivative of each term, applying rules such as the derivative rule for quadratic forms (14).

We then need to set this derivative equal to 0 to find the optimal weights in one go. Formally, this is called the "closed-form" solution.

$$\frac{\partial}{\partial \mathbf{w}} MSE = 0 \tag{16}$$

$$\iff \frac{1}{N}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}) = 0 \tag{17}$$

$$\iff \mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y} \tag{18}$$

$$\iff \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{19}$$

In other words this equation tells us what each weight in the weight vector should be, so that the model predicts in the best possible way, given the data we have fed it.

6 Inching towards the best weights

A second way to improve the weights we use, i.e. for our model to learn, is to nudge them towards the optimal weights. This is what "gradient descent" does. It calculates the gradient of the weight vector, i.e. the partial derivatives, and then multiplies this by a learning rate α to update weights.

$$\mathbf{w}' := \mathbf{w} - \alpha \cdot \frac{\partial}{\partial \mathbf{w}} MSE \tag{20}$$

In practice, weights are updated until some form of "convergence" is reached, which, for example, could be defined as the lack of significant loss reduction from one update compared to the previous one. In my code, I kept is simple and hard-coded the number of epochs.

7 Practicalities

The closed-form solution gives us the best possible weights in one go, which is neat. We pay, however, a price for this.

If gradient descent is coded in the right way, we will never multiply a matrix by another or its transpose. So at most, matrices are multiplied with vectors, or vectors with vectors, and the space complexity ends up being $O(D)$ with D being the number of features. Closed-form, on the other hand, has to invert a matrix to find the optimal weights, and textbooks tell us that depending on the algorithm used for this inversion, e.g. let's say Gaussian elimination, the space complexity is $O(D^2)$.

In terms of time complexity, gradient descent achieves $O(N \times D)$. This is because, for example, all the weights have to be multiplied by each feature for each row in the dataset. This is essentially a nested loop. The closed-form solution, however, again needs to do one thing that gradient descent does not need to: invert a matrix, which according to textbooks has a performance of D^3.

The question then becomes when the closed-form solution might become too expensive, assuming it is available at all. (It isn't for neural nets and logistic regression). For sure, the answer depends on what resources we have available. But let's do a back-of-the-envelope calculation. A common issue when training ML models is running out of VRAM. Often, people let GPUs run for several hours or days, and eat the time complexity, but running out of VRAM can kill training entirely.

Let's assume that we have $D = 100,000$ features, as is often the case in language modelling. Then we would have to store a $D \times D$ matrix for the closed-form algorithm, which assuming 4 bytes for a float64 (give or take - exact space seems to vary), would take up roughly 37 gigabytes. Some modern

GPUs could handle this, but there are other overheads that could quickly kill training. Gradient descent, on the other hand, would only need to store roughly six gigabytes for the equivalent calculation. In other words, closed-form scales badly.