Logistic Regression

Sia Sha

July 2025

1 Disclaimer

These notes exist only to clarify my own thinking. Should you find any errors, please write to me (mail at siavush dot co dot uk), so I can learn. I write these documents because reading something and understanding it are two different things. Also, understanding something and communicating it are two different things.

2 Introduction

The point of logistic regression is to classify/predict categorical outcomes, e.g. is a biopsy sample benign or malevolent for breast cancer? The way machines learn is always the same and so applies to logistic regression, too: we have a model that predicts/classifies something, we measure somehow how far off our predictions were, based on this deviation we teach the model to do better. So the recipe is forward pass $\rightarrow$ error $\rightarrow$ backward pass. Then, rinse and repeat.

3 Forward pass

If valid classification values are $y = \{0, 1\}$, a model such as linear regression would predict values outside of this set, and it would therefore not be a suitable model by itself. Instead, we need a model that is restricted to the set, and one solution is to the extend the linear regression model by sending its output through another function. The forward pass in logistic regression, therefore, becomes a two-step process. First, a dot product is calculated.

$$\mathbf{z} = \mathbf{X}\mathbf{w} \tag{1}$$

where $\mathbf{X}$ is a $N \times (D + 1)$ input matrix with N examples and $D + 1$ features, $\mathbf{w}$ is a column vector with shape $(D + 1) \times 1$, and $\mathbf{z}$ is $N \times 1$ column vector. The $+1$ exists to incorprorate a bias term, which in practice can be implemented by inserting a row of $1s$ into the feature matrix and adding one more weight to the weight vector.

Then, a prediction is calculated for certain input features by sending $\mathbf{z}$ through (2), which then restricts the range R_f as desired. The function (2) is called the sigmoid function.

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}) = \frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}} \tag{2}$$

4 Calculating error

As always, there are trade-offs with the error function we choose. The mean squared error would be a possible candidate, but it does not distinguish well between predictions that are off and those that are catastrophically off, penalising them similarly. Another choice, therefore, is to use the logistic cross-entropy error. The standard formula which most ML textbooks show is this one:

$$\mathcal{L}_{LCE}(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y}^T log(\hat{\mathbf{y}}) - (\mathbf{1} - \mathbf{y})^T log(1 - \hat{\mathbf{y}}) \tag{3}$$

However, there is a need to implement the loss in a numerically stable way. We can adjust (3) above to (10) below. Then, when implementing this in code, we can use the NumPy logsumexp function, which adds stability.

$$\mathcal{L}_{LCE}(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}_{LCE}(\mathbf{z}, \mathbf{y}) \tag{4}$$

$$= -\mathbf{y}^T log(\frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}}) - (\mathbf{1} - \mathbf{y})^T log(\mathbf{1} - \frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}}) \tag{5}$$

$$= -\mathbf{y}^T log(\frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}}) - (\mathbf{1} - \mathbf{y})^T log(\mathbf{1} - \frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}}) \tag{6}$$

$$= -\mathbf{y}^T \cdot \Big(log(\mathbf{1}) - log(\mathbf{1} + e^{-\mathbf{z}})\Big) - (\mathbf{1} - \mathbf{y})^T log(\frac{\mathbf{1} + e^{-\mathbf{z}} - \mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}}) \tag{7}$$

$$= \mathbf{y}^T log(\mathbf{1} + e^{-\mathbf{z}}) - (\mathbf{1} - \mathbf{y})^T log(\frac{e^{-\mathbf{z}} \cdot e^{\mathbf{z}}}{(\mathbf{1} + e^{-\mathbf{z}})e^{\mathbf{z}}}) \tag{8}$$

$$= \mathbf{y}^T log(\mathbf{1} + e^{-\mathbf{z}}) - (\mathbf{1} - \mathbf{y})^T \Big(log(\mathbf{1}) - log(1 + e^{\mathbf{z}})\Big) \tag{9}$$

$$= \mathbf{y}^T log(\mathbf{1} + e^{-\mathbf{z}}) + (\mathbf{1} - \mathbf{y})^T log(\mathbf{1} + e^{\mathbf{z}}) \tag{10}$$

An important note is that it is easy to make mistakes when doing algebraic manipulations. I had first messed up the above and had accidentally forgotten to drop the minus sign for the second term, i.e. (e^z, not e^{-z}). This tiny error could have caused catastrophic downstream effects, and even after a week of debugging I probably would have failed to figure out what is wrong. I guess two fail-safes are to work through every equation fully and thoroughly, and to cross reference the equations we use. Note, also, that in code it makes sense to use the average loss over the dataset. The above would be the total loss. I have left out $\frac{1}{n}$ to de-clutter the notation. This also applies to the gradient later calculated below.

5 L1 regularisation

L1 regularisation, also called "Lasso" regularisation, adds an extra term to the loss function. Specifically, it adds the L1 or Manhattan norm. Now the most important question would be why we would use this term and make our loss function more complicated. The answer is that the original Tibshirani paper shows that L1 regularisation improves the median mean squared error in crossvalidation. So L1 regularisation gives us better generalisation, or at least it does so under certain conditions.

The L1 loss would be:

$$\mathcal{L}_{L1} = \mathcal{L}_{LCE} + \lambda \|\mathbf{w}\|_1 \tag{11}$$

, where λ is a hyperparamter, which if set to 0, would just remove L1 regularisation. The mechanism by which L1 regularistion improves generalisation seems to be by pressure: it pressures weights to be 0. It's like exposing a herd to an infectious disease in order to thin it - and what survives are the strong. Now, it is interesting that adding the Manhattan norm to the loss itself doesn't do much. It's in the gradient that the L1 term takes its effect.

6 The gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{z}}$

So let's look at the gradient of the loss with respect to the weights: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{w}}$. To get this, we need to apply the chain rule. Let's calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{z}}$ first. Note that the we have to use the logistic

cross-entropy loss initially, not the L1 loss, as the L1 penalty term would disappear. But we can add it back in at the end.

$$\nabla_{\mathbf{z}}\mathcal{L}_{LCE} = \nabla_{\mathbf{z}}\Big(\mathbf{y}^T log(\mathbf{1}+e^{-\mathbf{z}}) + (\mathbf{1}-\mathbf{y})^T log(\mathbf{1}+e^{\mathbf{z}})\Big) \tag{12}$$

$$= \nabla_{\mathbf{z}}\mathbf{y}^T log(\mathbf{1}+e^{-\mathbf{z}}) + \nabla_{\mathbf{z}}(\mathbf{1}-\mathbf{y})^T log(\mathbf{1}+e^{\mathbf{z}}) \tag{13}$$

$$= -\mathbf{y} \odot \frac{e^{-\mathbf{z}}}{\mathbf{1}+e^{-\mathbf{z}}} + \nabla_{\mathbf{z}}(\mathbf{1}-\mathbf{y})^T log(\mathbf{1}+e^{\mathbf{z}}) \tag{14}$$

$$= -\mathbf{y} \odot \frac{e^{-\mathbf{z}}}{\mathbf{1}+e^{-\mathbf{z}}} + (\mathbf{1}-\mathbf{y}) \odot \frac{e^{\mathbf{z}}}{\mathbf{1}+e^{\mathbf{z}}} \tag{15}$$

$$= -\mathbf{y} \odot \frac{\mathbf{1}+e^{-\mathbf{z}}-\mathbf{1}}{\mathbf{1}+e^{-\mathbf{z}}} + (\mathbf{1}-\mathbf{y}) \odot \frac{e^{\mathbf{z}}}{\mathbf{1}+e^{\mathbf{z}}} \tag{16}$$

$$= -\mathbf{y} \odot (\mathbf{1}-\hat{\mathbf{y}}) + (\mathbf{1}-\mathbf{y}) \odot \frac{e^{\mathbf{z}} \cdot e^{-\mathbf{z}}}{(\mathbf{1}+e^{\mathbf{z}}) \cdot e^{-\mathbf{z}}} \tag{17}$$

$$= -\mathbf{y} \odot (\mathbf{1}-\hat{\mathbf{y}}) + (\mathbf{1}-\mathbf{y}) \odot \hat{\mathbf{y}} \tag{18}$$

$$= \hat{\mathbf{y}} - \mathbf{y} \tag{19}$$

There are several things worth noting at this stage.

1. The gradient is extremely simple in the end. Whoever first chose the logistic cross-entropy function, either by accident or design, created a loss function that works beautifully with the forward pass.
2. We stand on the shoulder of giants. Anyone new to ML will find several useful thoughts, properties, and derivations in books and lecture notes. Here, I am grateful to Grosse.
3. There are, however, gaps in ML education. To write efficient ML code, i.e. vectorised code, it makes sense to use vector/matrix notation when differentiating. Such notation, however, is rarely taught. In my education, I had seen no vector/matrix calculus, nor could I find the differentiation of (12) - (19) in that form on the web. I suffered from a skills issue, e.g. I didn't know what the derivative of a dot product is. So I spent several days reading books and articles, and one useful one was Parr and Howard.
4. Derivations are generally poorly annotated. Sometimes we do this to move quickly. But it isn't educational to do so for readers a lot of the time.

So let me list the most important steps for the above. In (12) and (13), we are distributing the gradient over the loss function terms. In (14) and (15), $\mathbf{y}^T$ and $(\mathbf{1}-\mathbf{y}^T)$ are constants. When using matrix multiplication in our scenario (a vector we are differentiating with respect to + a constant vector), then the result has to be a vector of partials, i.e a gradient. To illustrate, suppose

1. that we have a function $y = \mathbf{a}^T\mathbf{b}$ with two vectors,
2. that both vectors are of size $n \times 1$, and
3. that we are interested in $\frac{\partial y}{\partial \mathbf{b}}$

Then, the gradient is:

$$\nabla_{\mathbf{b}}\mathbf{y} = \begin{bmatrix} \frac{\partial}{\partial b_1}\Sigma a_j \cdot b_j \\ \frac{\partial}{\partial b_2}\Sigma a_j \cdot b_j \\ \vdots \\ \frac{\partial}{\partial b_m}\Sigma a_j \cdot b_j \end{bmatrix} = \begin{bmatrix} \Sigma\frac{\partial}{\partial b_1} a_i \cdot b_j \\ \Sigma\frac{\partial}{\partial b_2} a_i \cdot b_j \\ \vdots \\ \Sigma\frac{\partial}{\partial b_m} a_i \cdot b_j \end{bmatrix} \tag{20}$$

$$= \begin{bmatrix} \frac{\partial}{\partial b_1} a_1 \cdot b_1 \\ \frac{\partial}{\partial b_2} a_2 \cdot b_2 \\ \vdots \\ \frac{\partial}{\partial b_m} a_m \cdot b_m \end{bmatrix} = \begin{bmatrix} a_1 \cdot \frac{\partial}{\partial b_1} b_1 \\ a_2 \cdot \frac{\partial}{\partial b_2} b_2 \\ \vdots \\ a_m \cdot \frac{\partial}{\partial b_m} b_m \end{bmatrix} \tag{21}$$

$$= \begin{bmatrix} a_1 \cdot 1 \\ a_2 \cdot 1 \\ \vdots \\ a_m \cdot 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{22}$$

Given what a dot product is, the gradient has to be the sum of the multiplication of matching components in both vectors (20) with respect to each b_j. But since all products in that summation become 0 whenever $\frac{\partial}{\partial b_i}\Sigma a_j \cdot b_j$ is $j \neq i$, only really $a_i \cdot b_i$ remains and the Σ can be dropped. This means that the gradient of a dot product becomes essentially an elementwise multiplication between corresponding components, which is why in (14) the Hadamard product ($\odot$) replaces the dot product.

Returning to $\frac{\partial \mathcal{L}}{\partial \mathbf{z}}$, the rest of the differentiation becomes straightforward. Only two little tricks are worth noting: adding ± 1 in (16) and multiplying by $e^{-\mathbf{z}}$ in (17) in order to be able to substitute in $\hat{\mathbf{y}}$.

7 The gradient $\frac{\partial \mathbf{z}}{\partial \mathbf{w}}$

This one is a bit simpler than the previous derivative.

$$\nabla_{\mathbf{w}}\mathbf{z} = \frac{\partial}{\partial w_j}\mathbf{X}\mathbf{w} \tag{23}$$

$$= \frac{\partial}{\partial w_j} \begin{bmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \tag{24}$$

$$= \begin{bmatrix} \Sigma\frac{\partial}{\partial w_1} x_{1,j} \cdot w_j & \dots & \Sigma\frac{\partial}{\partial w_m} x_{1,j} \cdot w_j \\ \vdots & \ddots & \vdots \\ \Sigma\frac{\partial}{\partial w_1} x_{n,j} \cdot w_j & \dots & \Sigma\frac{\partial}{\partial w_m} x_{n,j} \cdot w_j \end{bmatrix} \tag{25}$$

$$= \begin{bmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{bmatrix} \tag{26}$$

$$= \mathbf{X} \tag{27}$$

8 The gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$

The gradient is as follows, applying the chain rule:

$$\nabla_{\mathbf{w}} \mathcal{L}_{LCE} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \tag{28}$$

$$= \mathbf{X}^T(\hat{\mathbf{y}} - \mathbf{y}) \tag{29}$$

Again, I have not been able to find the differentiation of the above gradient in vector form anywhere, so I differentiated from first principles to the best of my ability. It does mirror the derivative in the scalar notation (a good sign) as well as vectorised code on the internet. Note that $\mathbf{X}$ needs to be transposed and multiplied from the right so the dimensions match. I assumed in my notation that all vectors by default are column vectors. Finally, remember that this in code needs to be multiplied by $1/N$ still.

9 The Gradient of the regularisation term

We still need to add the derivative to the regularisation term to have the final gradient. Let's look at some initial steps.

$$\nabla_{\mathbf{w}} \mathcal{L}_{L1} = \frac{\partial \mathcal{L}_{L1}}{\partial w_j} \lambda \|\mathbf{w}\|_1 \tag{30}$$

$$= \lambda \begin{bmatrix} \frac{\partial \mathcal{L}_{L1}}{\partial w_1} \Sigma |w_j| \\ \frac{\partial \mathcal{L}_{L1}}{\partial w_2} \Sigma |w_j| \\ \vdots \\ \frac{\partial \mathcal{L}_{L1}}{\partial w_m} \Sigma |w_j| \end{bmatrix} \tag{31}$$

$$= \lambda \begin{bmatrix} \Sigma \frac{\partial \mathcal{L}_{L1}}{\partial w_1} |w_j| \\ \Sigma \frac{\partial \mathcal{L}_{L1}}{\partial w_2} |w_j| \\ \vdots \\ \Sigma \frac{\partial \mathcal{L}_{L1}}{\partial w_m} |w_j| \end{bmatrix} \tag{32}$$

$$= \lambda \begin{bmatrix} \frac{\partial \mathcal{L}_{L1}}{\partial w_1} |w_1| \\ \frac{\partial \mathcal{L}_{L1}}{\partial w_2} |w_2| \\ \vdots \\ \frac{\partial \mathcal{L}_{L1}}{\partial w_m} |w_3| \end{bmatrix} \tag{33}$$

Essentially, the derivative therefore becomes with respect to each weight: $\frac{\partial \mathcal{L}1}{\partial w_j}|w_j|$. If we then really think about what it means to take the absolute value of scalar, we can rewrite this as:

$$\frac{\partial \mathcal{L}1}{\partial w_j}|w_j| = \frac{\partial \mathcal{L}1}{\partial w_i}\sqrt{(w_j)^2} \tag{34}$$

$$= \frac{1}{2}(w_j^2)^{-\frac{1}{2}} \cdot 2w_j \tag{35}$$

$$= \frac{2w_j}{2\sqrt{w_j^2}} \tag{36}$$

$$= \frac{w_j}{|w_j|} \tag{37}$$

$$\tag{38}$$

This means that the final gradient is:

$$\nabla_{\mathbf{w}}\mathcal{L}_{L1} = \lambda \begin{bmatrix} \frac{w_1}{|w_1|} \\ \frac{w_2}{|w_2|} \\ \vdots \\ \frac{w_m}{|w_m|} \end{bmatrix} \tag{39}$$

The above applies for all $w_j \neq 0$ due to division by 0. Where $w_j = 0$, the derivative is simply 0. The obvious questions is why 0? The issue is that the function is not differentiable at $w_j = 0$. So we need to plug in a value that serves our needs and that still leads to convergence. The choice of 0 is nice, because weights that are already 0 (the ones that were thinned out) are not pushed away from 0, thus "staying dead".

10 How does regularisation work exactly?

So here is then, finally, how exactly L1 regularisation works. The overall gradient tells us the direction and magnitude by which we should adjust each weight in order to improve the loss. The L1 term just adds a wee bit extra pressure (± 1) to the regular gradient, and it means that we are a little bit overcorrecting weights. The magnitude of this overcorrection depends on λ, and can be small or large. The effect of this added extra pressure is that important weights, i.e. those that have a large effect on the loss, "will fight for their survival". Unimportant weights, on the other hand, will struggle to fight and "succumb to the pathogen", diminishing to near-zero or zero over time.

Note 1: back in the day researchers had to carefully curate their variables and only include the most relevant ones in their models. Given the constant pressure towards 0 for weights, these days, researchers throw all their variables into the model, and let the Lasso choose which ones matter.

Note 2: people may claim all sorts of things about L1 regularisation, maybe even things that the original paper did not claim. The question is whether there is evidence for it. Some may say that L1 helps with the bias of estimators. Maybe, but I haven't seen evidence for it (not that I specifically searched for it, though).

Note 3: L1 regularisation intentionally underfits the training data a bit, which then improves generalisation.