Logistic Regression

Logistic regression, described in this note, is a standard work-horse of practical machine learning. It's possible to get somewhere with many applied problems by making some binary decisions. Working out how to make these decisions with logistic regression is an important baseline, and could even be where you stop.

1 Transforming the output

We've previously seen that we can attempt to fit binary labels $y \in \{0, 1\}$ with straightforward linear regression models. We derived that the ideal function is $f(\mathbf{x}) = P(y = 1 \,|\, \mathbf{x})$, the probability of being in class 1. However, linear functions we fit are likely to predict outputs outside the range $[0, 1]$ for some test inputs. We will now force our function to lie in the desired $[0, 1]$ range by transforming a linear function with a logistic sigmoid:

$$f(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}\,. \tag{1}$$

As with linear regression, we can replace our input features $\mathbf{x}$ with a vector of basis function values $\boldsymbol{\phi}(\mathbf{x})$, but we won't clutter the notation with this detail. Wherever you see $\mathbf{x}$, know that you can replace this input vector with a version that has been transformed in any way you like.

2 Loss function

As before, we wish to fit the function to match the training data closely, If the labels are zero and one, $y \in \{0, 1\}$, we *could* minimize the square loss[1]

$$\sum_{n=1}^{N} \left(y^{(n)} - f(\mathbf{x}^{(n)}; \mathbf{w})\right)^2. \tag{2}$$

However, the standard *Logistic Regression* model uses the interpretation of the function as a probability, $f(\mathbf{x}; \mathbf{w}) = P(y = 1 \,|\, \mathbf{x}, \mathbf{w})$, more directly. Maximum likelihood fitting of this model maximizes the probability of the data:

$$L(\mathbf{w}) = \prod_{n=1}^{N} P(y^{(n)} \,|\, \mathbf{x}^{(n)}, \mathbf{w}), \tag{3}$$

for the model with parameters $\mathbf{w}$. Equivalently, we minimize the negative log-probability of the training labels, which for this model can be written as:

$$\text{NLL} = -\log L(\mathbf{w}) = -\sum_{n=1}^{N} \log \left[\sigma(\mathbf{w}^\top \mathbf{x}^{(n)})^{y^{(n)}} (1 - \sigma(\mathbf{w}^\top \mathbf{x}^{(n)}))^{1-y^{(n)}}\right], \tag{4}$$

or

$$\text{NLL} = -\sum_{n:y^{(n)}=1} \log \sigma(\mathbf{w}^\top \mathbf{x}^{(n)}) - \sum_{n:y^{(n)}=0} \log(1 - \sigma(\mathbf{w}^\top \mathbf{x}^{(n)})). \tag{5}$$

There is a trick to write the cost function more compactly. We transform the labels to be $z^{(n)} \in \{-1, +1\}$ where $z^{(n)} = (2y^{(n)} - 1)$, and noticing $\sigma(-a) = 1 - \sigma(a)$, we can write:

$$\text{NLL} = -\sum_{n=1}^{N} \log \sigma(z^{(n)} \mathbf{w}^\top \mathbf{x}^{(n)}). \tag{6}$$

1. A NeurIPS 2019 paper still found space to discuss this loss for classification.

As before, the cost function can have a regularizer added to it to discourage extreme weights.

Maximum likelihood estimation has some good statistical properties. In particular, asymptotically (for lots of data) it is the most efficient estimator. Although the loss can be extreme where confident wrong predictions are made, which could mean that outliers cause more problems than with the square loss approach.

[The website version of this note has a question here.]

3 Gradients

The final required ingredient is the gradient vector $\nabla_{\mathbf{w}}\text{NLL}$. A gradient-based optimizer can then find the weights that minimize our cost.

The derivative of the logistic sigmoid is as follows:

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a)), \tag{7}$$

which tends to zero at the asymptotes $a \to \pm\infty$.

I like to derive the derivatives using the form of the cost function in (6), because it's shorter, although I think most books use (4). We'll get an equivalent answer. For brevity, I'll use $\sigma_n = \sigma(z^{(n)}\mathbf{w}^\top\mathbf{x}^{(n)})$. We then apply the chain rule:

$$\nabla_{\mathbf{w}}\text{NLL} = -\sum_{n=1}^{N} \nabla_{\mathbf{w}} \log \sigma_n = -\sum_{n=1}^{N} \frac{1}{\sigma_n}\nabla_{\mathbf{w}}\sigma_n = -\sum_{n=1}^{N} \frac{1}{\sigma_n}\sigma_n(1-\sigma_n)\nabla_{\mathbf{w}} z^{(n)}\mathbf{w}^\top\mathbf{x}^{(n)}, \tag{8}$$

$$= -\sum_{n=1}^{N} (1-\sigma_n) z^{(n)}\mathbf{x}^{(n)}. \tag{9}$$

Interpretation: σ_n is the probability assigned to the correct training label. So when the classifier is confident and correct on an example, it contributes little to the gradient. Stochastic gradient descent will improve the examples the classifier gets wrong, or is less confident about, by pushing the weights parallel to the direction of the corresponding input.

4 Some things to know about gradients

Whenever we can compute a differentiable cost function, it should always be possible to compute *all* of the derivatives at once in a similar number of operations to one function evaluation. That's an amazing result from the old field of *automatic differentiation*. (Caveat it might take a lot of memory.) So if your derivates are orders of magnitude more expensive than your cost function, you are probably doing something wrong.

Despite a long history, few people use fully automatic differentiation in machine learning. Many machine learning frameworks (such as PyTorch or Tensor Flow) will do most of the work for you. But some people[2] still sometimes need to do some work by hand so that those tools can work on all the models that we might build.

Whether you are differentiating by hand, or writing a compiler to compute derivatives, you need to test your code. Derivatives are easily checked by *finite differences*:

$$\frac{\partial f(w)}{\partial w} \approx \frac{f(w+\epsilon/2) - f(w-\epsilon/2)}{\epsilon}, \tag{10}$$

and so should always be checked. Unless the weights are extreme, I'd normally set $\epsilon = 10^{-5}$. You have to perturb each parameter in turn to evaluate one element of a gradient vector $\nabla_{\mathbf{w}} f(\mathbf{w})$. Therefore, for D-dimensional vectors of derivatives, the computational cost of finite differences scales D times worse than well-written derivative code, as well as being less accurate. Finite differences are a useful check, but not for use in production.

[The website version of this note has a question here.]

2. Your lecturer is one of them: `https://arxiv.org/abs/1602.07527`.

5 Further Reading

All machine learning textbooks should have a treatment of logistic regression. You could read an alternative treatment to this note in Bishop 4.3.2, or Barber 17.4.1 and 17.4.4. Or Murphy: quick introduction Section 1.4.6, then Chapter 8, which has an in depth treatment beyond this note.

Tom Minka has a review of alternative batch optimizers for logistic regression. Stochastic gradient methods were less popular then, and were not considered.

For a large-scale practical tool that uses stochastic optimization, check out Vowpal Wabbit. Its framework includes support for logistic regression. It has various tricks to train *fast*. It can cope with data, like large-scale text corpora, where you might not know what you want your features to be until you start streaming data.

We don't have to transform a linear function with the logistic sigmoid. We could instead create a 'probit' model, which uses a Gaussian's cumulative density function (cdf) instead of the logistic sigmoid. We can also transform the function to model other data types. For example, count data can be modeled by using the underlying linear function to set the log-rate of a Poisson distribution. These alternatives can be unified as "Generalized Linear Models" (GLMs). R has a widely used glm library.

There is a cool trick with complex numbers to evaluate derivatives to machine precision, which I'd like to share: complex step differentiation (Mathworks blog). It is no faster than finite differences though, so also shouldn't be used, except as a check.