Linear Classification with Logistic Regression

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

When discussing linear regression, we examined two different points of view that often led to similar algorithms: one based on constructing and minimizing a loss function, and the other based on maximizing the likelihood. Classification has a similar set of parallel viewpoints and algorithms, but we'll start with the probabilistic view.

In probabilistic linear regression, we studied the idea that there was some generating procedure that took in an input, produced an idealized function and then added noise to it. We examined, in particular, the case where that noise was zero-mean Gaussian noise. In probabilistic classification we will take a similar view, except that a Gaussian distribution will not make sense because the data will now be binary rather than real-valued. Our go-to distribution for binary data is the Bernoulli, which is just the biased coin flip. The outcome can take the value 0 or 1 and there is a parameter $\theta \in [0, 1]$ that is the mean of the distribution. The probability mass function (PMF) is

$$\Pr(y \mid \theta) = \theta^y (1-\theta)^{1-y} \qquad \text{(Bernoulli PMF)}\,. \tag{1}$$

This PMF might look strangely complicated for coin flips if you haven't seen it before, but all that's going on here is that it's using the fact that $z^0 = 1$ and $z^1 = z$ as a kind of trick to slice out the right values. What we're going to do to turn this into a model for supervised binary classification is to say that θ is a function of the input $\boldsymbol{x}$. We can't directly use the function $\boldsymbol{w}^\mathsf{T}\boldsymbol{x}$ because that will produce values less than 0 and greater than 1. To address this, we use a function that transform $\boldsymbol{w}^\mathsf{T}\boldsymbol{x}$ into $[0, 1]$. There are various choices we could make for such a function, but the most common thing is to choose the *logistic function*:

$$\sigma(z) = \frac{\exp\{z\}}{1+\exp\{z\}} = \frac{1}{1+\exp\{-z\}}\,. \tag{2}$$

This function is shown in Figure 1 where you can see that this is an example of a *sigmoid* ("s-shaped") function. We often use $\sigma(\cdot)$ to denote this function.

Putting these pieces together, we can construct a model that takes in a location $\boldsymbol{x}$ (and weights $\boldsymbol{w}$) and produces a Bernoulli distribution:

$$\Pr(y \mid \boldsymbol{x}, \boldsymbol{w}) = \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x})^y \, (1 - \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}))^{1-y}\,. \tag{3}$$

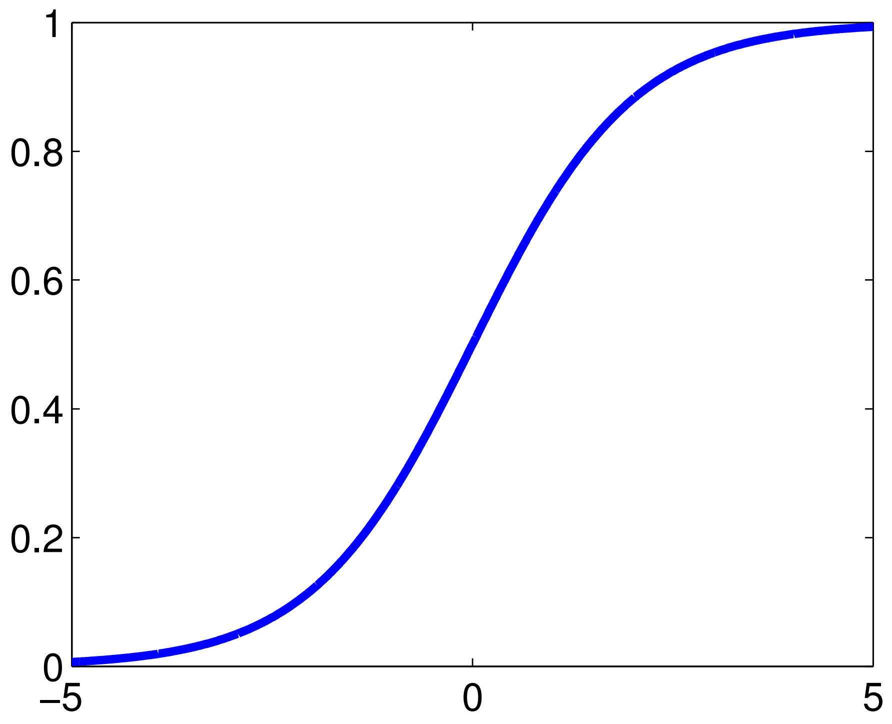

Figure 1: The logistic function $f(z) = 1/(1 + \exp\{-z\})$.

The actual problem we want to solve, however, is to find the maximum likelihood estimate of $\boldsymbol{w}$ after seeing N data $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ where $\boldsymbol{x}_n \in \mathbb{R}^D$ and $y_n \in \{0, 1\}$. We are taking these to be independent Bernoulli distributions, conditioned on $\boldsymbol{w}$ and the $\boldsymbol{x}_n$, so the likelihood is a product:

$$\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, \boldsymbol{w}) = \prod_{n=1}^N \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)^{y_n} (1 - \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n))^{1-y_n} \,. \tag{4}$$

This is the function that we will want to maximize with respect to $\boldsymbol{w}$, and as in the linear regression case we'll want to take the log first to avoid numeric difficulties due to products of small numbers:

$$\boldsymbol{w}^{\mathsf{MLE}} = \arg\max_{\boldsymbol{w}} \left\{ \sum_{n=1}^N y_n \log \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) + (1 - y_n) \log(1 - \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)) \,. \right\} \tag{5}$$

Even though this objective function is concave in $\boldsymbol{w}$, it is not possible to minimize it directly be setting the gradient to zero and solving for $\boldsymbol{w}$ as we did with linear regression. Instead, we'll have to minimize this the hard way, using gradient ascent.

Gradient Ascent/Descent

The idea of gradient descent is to solve the problem

$$\min_z f(z)\,, \tag{6}$$

in a situation where we have access to the gradient $\nabla_z f(z)$ but limited additional information. (Gradient ascent is the same idea, but where we're maximizing and we flip the signs of everything. Here I'll frame everything in terms of minimization.) Gradient descent observes that the negative gradient points in the direction of steepest descent. If we take a small enough step in that direction, then we're very likely to go "downhill" and find a z that reduces the value of $f(z)$:

$$z^{(t+1)} \leftarrow z^{(t)} - \alpha \nabla_z f(z^{(t)}), \tag{7}$$

where we start at some arbitrary (perhaps random) initialization $z^{(0)}$. The constant $\alpha > 0$ must be simultaneously small enough that we're tending to move downhill, while large enough that we make progress. Iteratively taking such steps will send us toward a critical point (a place where the gradient is zero) and in convex problems this critical point will be the global minimum. In non-convex problems we often simply cross our fingers and hope that the critical point we converge to is a minimum that is not too bad.

Newton's Method Setting α can be difficult in practice, and it often needs to vary over the course of the optimization in order to achieve a good solution. Moreover, as can be seen from the zig-zagging pathology, going directly downhill may not be the best thing to do if the local shape of the function is stretched out in some directions and compressed in others. Newton's method is one important example of a *second order* optimization method. Roughly speaking, the order of an optimization approach refers to the number of derivatives used, so gradient descent is a first order method, while a second order method would use the Hessian matrix in some form. The idea of Newton's method is to assume that the function we are trying to minimize is approximately quadratic in the immediate vicinity of our current iterate $f(z)$. We can estimate that quadratic using a Taylor expansion around the current point $z^{(t)}$:

$$f(z) \approx f(z^{(t)}) + (z - z^{(t)})^{\mathsf{T}} \nabla_z f(z^{(t)}) + \frac{1}{2}(z - z^{(t)})^{\mathsf{T}} \mathcal{H}_z[f(z^{(t)})](z - z^{(t)}), \tag{8}$$

where $\mathcal{H}_z[f(\cdot)]$ is the Hessian of $f(\cdot)$ with respect to z. If this were the true function, then we could actually compute the minimum exactly by taking the gradient, setting it to zero:

$$\begin{align} &\nabla_z \left\{ f(z^{(t)}) + (z - z^{(t)})^{\mathsf{T}} \nabla_z f(z^{(t)}) + \frac{1}{2}(z - z^{(t)})^{\mathsf{T}} \mathcal{H}_z[f(z^{(t)})](z - z^{(t)}) \right\} \tag{9} \\ &= \nabla_z f(z^{(t)}) + \mathcal{H}_z[f(z^{(t)})](z - z^{(t)}) = 0 \tag{10} \end{align}$$

and then solving for z:

$$\begin{align} \nabla_z f(z^{(t)}) + \mathcal{H}_z[f(z^{(t)}]z - \mathcal{H}_z[f(z^{(t)})]z^{(t)} &= 0 \tag{11} \\ \mathcal{H}_z[f(z^{(t)}]z &= \mathcal{H}_z[f(z^{(t)})]z^{(t)} - \nabla_z f(z^{(t)}) \tag{12} \\ z &= \mathcal{H}_z[f(z^{(t)}]^{-1}(\mathcal{H}_z[f(z^{(t)})]z^{(t)} - \nabla_z f(z^{(t)})) \tag{13} \\ &= z^{(t)} - \mathcal{H}_z[f(z^{(t)}]^{-1} \nabla_z f(z^{(t)}). \tag{14} \end{align}$$

If we imagine then at each step of the optimization saying "assume $f(\cdot)$ is locally quadratic and jump to where the minimum should be" then you get the update:

$$z^{(t+1)} \leftarrow z^{(t)} - \mathcal{H}_z[f(z^{(t)}]^{-1}\nabla_z f(z^{(t)})\,. \tag{15}$$

This is exactly like the gradient descent update but rather than scale the gradient with a constant α, we use it to solve a linear system with the Hessian. There are a huge number of ways this can go wrong and so there is a large literature on variations and tweaks to improve things. For example, one may not have direct access to the Hessian but can only compute Hessian-vector products; the Hessian may be too big and so you don't want to represent it at all, much less solve a linear system with it; you may want to add a learning rate anyway rather than try to jump all the way to the solution; your Hessian may not be positive definite and so this method may tell you to jump to infinity. In the current moment of machine learning, where people seem to care the most about optimizing large neural networks, second order methods seem to offer no practical improvement at all over first order methods, or at least not enough to justify their complexity and computational cost.

Stochastic Gradient Descent The workhorse of machine learning at the moment is *stochastic* gradient descent (SGD). In SGD, we don't have access to the true gradient but only to a noisy version of it. It turns out that if the noise isn't too bad, and you decay the learning rate over time, then you will still converge to a solution. The way in which this is most helpful is in tackling large data sets with gradient descent: the true gradient of the training loss will be an average over all of the data, but we can often estimate it well using a small subset ("mini-batch") of the data. This will be an unbiased estimate and so things are still likely to work. It additionally seems to be the case that the noise arising from stochastic gradient descent for deep neural networks actually helps them generalize by somehow avoiding poor local minima in the training loss. That is, some early theoretical evidence and much empirical evidence indicates that the noisy gradient introduces an implicit regularization into the model that helps prevent overfitting.

SGD for Logistic Regression

We now return to the problem specified by Eqn. 5 and examine the gradient arising from a single one of the data:

$$\nabla_{\boldsymbol{w}} \left\{ y_n \log \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) + (1-y_n)\log(1-\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)) \right\}\,. \tag{16}$$

We're going to perform gradient descent by performing updates that subtract the negative of the gradient, i.e., by adding the gradient. We'll then make this a stochastic method by choosing data uniformly at random rather than summing over the entire data set.

First, there are two good identities to know about the logistic function:

$$1-\sigma(z) = 1 - \frac{\exp\{z\}}{1+\exp\{z\}} = \frac{1+\exp\{z\}}{1+\exp\{z\}} - \frac{\exp\{z\}}{1+\exp\{z\}} = \frac{1}{1+\exp\{z\}} = \sigma(-z) \tag{17}$$

and

$$\frac{d}{dz}\sigma(z) = \frac{d}{dz}(1+\exp\{-z\})^{-1} = \frac{\exp\{-z\}}{(1+\exp\{-z\})^2} = \frac{\exp\{-z\}}{1+\exp\{-z\}}\frac{1}{1+\exp\{-z\}} \tag{18}$$

$$= \sigma(-z)\sigma(z) = (1-\sigma(z))\sigma(z)\,. \tag{19}$$

We can use these to get an intuitive form for the gradient:

$$\nabla_{\boldsymbol{w}}\left\{y_n \log\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) + (1-y_n)\log(1-\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n))\right\} \tag{20}$$

$$= y_n\frac{1}{\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)}\nabla_{\boldsymbol{w}}\{\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)\} + (1-y_n)\frac{1}{\sigma(-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)}\nabla_{\boldsymbol{w}}\{\sigma(-\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)\} \tag{21}$$

$$= y_n\boldsymbol{x}_n(1-\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)) - (1-y_n)\boldsymbol{x}_n\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) \tag{22}$$

$$= y_n\boldsymbol{x}_n - y_n\boldsymbol{x}_n\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) - \boldsymbol{x}_n\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) + y_n\boldsymbol{x}_n\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) \tag{23}$$

$$= y_n\boldsymbol{x}_n - \boldsymbol{x}_n\sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) \tag{24}$$

$$= \boldsymbol{x}_n(y_n - \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n))\,. \tag{25}$$

The single-example gradient can then be used to form an unbiased estimate of the true (full-data) gradient by sampling n uniformly at random from $1,\ldots,N$ and then using the nth datum to perform the update:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \alpha\boldsymbol{x}_n(y_n - \sigma((\boldsymbol{w}^{(t)})^\mathsf{T}\boldsymbol{x}_n))\,. \tag{26}$$

Remarkably, this is actually the same rule we identified for gradient descent for least squares regression in that it takes a step proportional to the error, weighted by the input features. It is almost exactly what we saw from the perceptron learning rule, except using the sigmoid function rather than the sign function.

Linear Separability and Regularization In general, linear separability is a good thing for a binary classification problem. It means the problem is easy in some sense, and simple algorithms like the perceptron learning rule will work. However, it creates a pathology for unregularized logistic regression. Consider the fact that the decision boundary in a linear classifier is independent of the scale of the parameters. You can see this by recalling that the decision boundary is the set $\{\boldsymbol{x} : \boldsymbol{w}^\mathsf{T}\boldsymbol{x} = 0\}$ and that this set isn't changed if we multiply $\boldsymbol{w}$ by some constant c. For a given decision boundary, however, the scale does effect the likelihood in logistic regression by causing the logistic function to become more steep. This is probably an obvious statement, but just in case: you can see this steepness by thinking about the derivative of $\sigma(z)$ evaluated at $z = 0$ versus the derivative of $\sigma(10z)$ at $z = 0$. The derivatives are $\sigma(z)(1-\sigma(z))$ and $10\sigma(z)(1-\sigma(z))$, respectively, and so that linear regime in the middle is ten times steeper when the input is scaled by a factor of ten.

If we currently have a decision boundary such that the data are all correctly classified, then increasing the scale of the weights will push the predictions further towards their correct answers. Imagine that we have a set of weights $\hat{\boldsymbol{w}}$ with unit norm, i.e., $||\hat{\boldsymbol{w}}|| = 1$ for whatever norm you want. We construct a logistic regression classifier with weights $\boldsymbol{w} = c\hat{\boldsymbol{w}}$ and seek only to fit the constant $c > 0$ to the data. Recall that changing c does not move the decision boundary for the classifier. We take the nth example and examine the derivative of its log likelihood with respect to c:

$$\frac{\partial}{\partial c}\left\{y_n \log \sigma(c\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n) + (1-y_n)\log(1-\sigma(c\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n))\right\} = \hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n(y_n - \sigma(c\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n))\,. \tag{27}$$

If $y_n = 0$ then $(y_n - \sigma(c\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n)) < 0$ and if $y_n = 1$ then $(y_n - \sigma(c\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n)) > 0$. Note also that due to the fixed decision boundary, if $y_n = 0$ is classified correctly then $\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n < 0$ and is positive otherwise. Similarly if $y_n = 1$ is classified correctly, then $\hat{\boldsymbol{w}}^{\mathsf{T}}\boldsymbol{x}_n > 0$ and is negative otherwise. Thus the derivative of the log likelihood with respect to c is always positive for an example that $\hat{\boldsymbol{w}}$ classifies correctly. If the data are linearly separable, then there exists a $\hat{\boldsymbol{w}}$ such that all of the data have log likelihoods with positive derivatives with respect to c. In that situation, gradient ascent on c would cause it to grow without bound. This essentially drives the sigmoid function to be sharper and sharper until it becomes a Heaviside step function. This is a kind of overfitting: the model is becoming perfectly confident about the data and using very large weights to achieve it.

We have already learned a solution to this problem: regularize the weights. A common thing to do is to use the same squared L^2 norm that we used in ridge regression: essentially saying as before that we are going to find the MAP with a Gaussian prior on the weights.

$$\boldsymbol{w}^{\mathsf{MAP}} = \arg\max_{\boldsymbol{w}} \left\{\log \Pr(\{y_n\}_{n=1}^N \,|\, \{\boldsymbol{x}_n\}_{n=1}^N, \boldsymbol{w}) - \frac{\lambda}{2}||\boldsymbol{w}||_2^2\right\}\,. \tag{28}$$

The gradient of the resulting objective is then

$$\nabla_{\boldsymbol{w}}\left\{\sum_{n=1}^N y_n \log \sigma(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n) + (1-y_n)\log(1-\sigma(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n)) - \frac{\lambda}{2}\boldsymbol{w}^{\mathsf{T}}\boldsymbol{w}\right\} \tag{29}$$

$$= \sum_{n=1}^N \boldsymbol{x}_n(y_n - \sigma(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n)) - \lambda\boldsymbol{w}\,. \tag{30}$$

The constant λ now has a scale relative to N, so we can either make our single example stochastic updates scale up by a factor of N or scale λ down by a factor of N. Since α and λ are arbitrary constants, this doesn't have a practical effect on the algorithm. However, using the latter adjustment results in small addition to the previous stochastic gradient descent update rule:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \alpha\left(\boldsymbol{x}_n(y_n - \sigma((\boldsymbol{w}^{(t)})^{\mathsf{T}}\boldsymbol{x}_n)) - \frac{\lambda}{N}\boldsymbol{w}^{(t)}\right)\,, \tag{31}$$

which we can rewrite as:

$$\boldsymbol{w}^{(t+1)} \leftarrow (1 - \frac{\alpha\lambda}{N})\boldsymbol{w}^{(t)} + \alpha\boldsymbol{x}_n(y_n - \sigma((\boldsymbol{w}^{(t)})^{\mathsf{T}}\boldsymbol{x}_n))\,. \tag{32}$$

This shows why machine learning researchers (and neural network researchers in particular) often refer to L^2 regularization as "weight decay". In the gradient ascent update rules, this regularization term introduces a "decay toward zero then add the gradient" dynamic.

Beyond Binary Classification

Unlike some binary classification approaches, logistic regression generalizes naturally to $K > 2$ classes. This is essentially because the Bernoulli (binomial) distribution generalizes directly to the categorical (multinomial) distribution. The parameter is an element of the $K - 1$ simplex, i.e., $\boldsymbol{\theta} \in \mathbb{R}^K$, where $\theta_k > 0$ and $\sum_{k=1}^K \theta_k = 1$. Even though $\boldsymbol{\theta}$ has K dimensions, it only has $K - 1$ degrees of freedom, since it must sum to one.

For the data, rather than imagining that our labels are $y_n \in \{0, 1\}$ we now imagine that they are $\boldsymbol{y}_n \in \{0, 1\}^K$ subject to the constraint that $\sum_{k=1}^K y_{n,k} = 1$. This is what we refer to as a "one-hot coding": a binary vector with as many dimensions as classes and all zeros except for a one in the dimension of the label for the example. We can then write an equivalent to Eqn. 1 as

$$\Pr(\boldsymbol{y} \mid \boldsymbol{\theta}) = \prod_{k=1}^{K} \theta_k^{y_k} \,. \tag{33}$$

As in binary logistic regression, we have to find a way to map our inputs $\boldsymbol{x} \in \mathbb{R}^D$ into the vector $\boldsymbol{\theta}$. For $K > 2$, we'll have K weight vectors $\boldsymbol{w}_k \in \mathbb{R}^D$ and we will compute the inner product of $\boldsymbol{x}$ with each of them. After that, we will exponentiate them and then divide by the total across the classes:

$$\theta_k = \frac{\exp\{\boldsymbol{x}^\mathsf{T}\boldsymbol{w}_k\}}{\sum_{k'=1}^K \exp\{\boldsymbol{x}^\mathsf{T}\boldsymbol{w}_{k'}\}} \,. \tag{34}$$

This exponentiate-and-normalize is often called a *softmax* and it ensures that each of the values is non-negative and sums to one, as we require for $\boldsymbol{\theta}$. Combining together Eqns. 33 and 34 we can write a "softmax regression" likelihood:

$$\Pr(\boldsymbol{y} \mid \boldsymbol{x}, \{\boldsymbol{w}_k\}_{k=1}^K) = \prod_{k=1}^{K} \left(\frac{\exp\{\boldsymbol{x}^\mathsf{T}\boldsymbol{w}_k\}}{\sum_{k'=1}^K \exp\{\boldsymbol{x}^\mathsf{T}\boldsymbol{w}_{k'}\}} \right)^{y_k} \tag{35}$$

With this likelihood in hand, we can write the optimization problem for maximizing the log likelihood after seeing N data:

$$\{\boldsymbol{w}_k^{\mathsf{MLE}}\}_{k=1}^K \operatorname*{arg\,max}_{\{\boldsymbol{w}_k\}_{k=1}^K} \left\{ \sum_{n=1}^N \left(\sum_{k=1}^K y_{n,k} \boldsymbol{x}_n^\mathsf{T} \boldsymbol{w}_k \right) - \log \sum_{k=1}^K \exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_k\} \right\} \,. \tag{36}$$

As in binary logistic regression, we maximize this by taking the gradient and performing (stochastic) gradient ascent:

$$\nabla_{\boldsymbol{w}_k} \left\{ \sum_{n=1}^N \left(\sum_{k=1}^K y_{n,k} \boldsymbol{x}_n^\mathsf{T} \boldsymbol{w}_k \right) - \log \sum_{k=1}^K \exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_k\} \right\} = \sum_{n=1}^N y_{n,k}\boldsymbol{x}_n - \frac{\exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_k\}}{\sum_{k'=1}^K \exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_{k'}\}} \boldsymbol{x}_n \tag{37}$$

$$= \sum_{n=1}^N \boldsymbol{x}_n \left(y_{n,k} - \frac{\exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_k\}}{\sum_{k'=1}^K \exp\{\boldsymbol{x}_n^\mathsf{T}\boldsymbol{w}_{k'}\}} \right) \,. \tag{38}$$

This is satisfying as a fairly direct analog to Eqn. 25: the inputs weighted by the difference between the true label and the prediction.

An Aside: Computing Log-Sum-Exp The log-of-sum-of-exponentials term in Eqn. 36 comes up a lot in machine learning and it is annoying because it is numerically prone to underflow and overflow. Let's look at a simplified version for a vector $\boldsymbol{z} \in^J$:

$$\log \sum_{j=1}^{J} \exp\{z_j\} \qquad \text{(Log-Sum-Exp)} \qquad (39)$$

Imagine that one entry in $\boldsymbol{z}$ is much larger than the others. In this case, the value of the log-sum-exp will essentially just be that large entry in $\boldsymbol{z}$. However, exponentiating a large floating point number may overflow and give you `inf`. Taking the log of `inf` will still be `inf` (or `NaN`), which is not what you want. We can tweak things to be better behaved, however, by introducing an arbitrary constant c. Note that we can roll a constant into the log-sum-exp without changing its value:

$$\log \sum_{j=1}^{J} \exp\{z_j\} = c + \log\{\exp\{-c\}\} + \log \sum_{j=1}^{J} \exp\{z_j\} = c + \log \left\{ \exp\{-c\} \sum_{j=1}^{J} \exp\{z_j\} \right\} \qquad (40)$$

$$= c + \log \sum_{j=1}^{J} \exp\{z_j - c\} \qquad (41)$$

If we make $c = \max_j z_j$ then now the largest thing we're taking an exponential of is zero. All of the other values are less than or equal to zero, so we will not get underflow. The values might be large and negative, but this is tolerable because, in floating point, underflow of the exponential function just gives zero. In the worst case, after exponentiation everything but the big value becomes zero, and the big value becomes one. Then the log term goes away and the entire quantity is just $c = \max_j z_j$, which is essentially the correct answer.

Generalized Linear Models

Logistic regression is a special case of a popular and important class of statistical models called *generalized linear models* (GLMs). The GLM framework allows one to model different kinds of label spaces using this same recipe of linear function, nonlinear transformation, and likelihood. Note that in linear regression, binary logistic regression, and softmax regression, we were using a linear function of $\boldsymbol{x}$ to parameterize the mean of the distribution on the output. The GLM frames this in a slightly different way than we have here, by calling the inverse transformation a *link function*, but the concept is essentially the same. A couple of common examples of GLM likelihoods are the Poisson, where the labels are non-negative integers:

$$\Pr(y_n \,|\, \lambda_n) = \frac{\lambda_n^{y_n} \exp\{\lambda_n\}}{y_n!} \qquad \lambda_n = \exp\{\boldsymbol{w}^\mathsf{T} \boldsymbol{x}_n\} \qquad (42)$$

and similarly one could construct an exponential distribution regression model on the positive reals:

$$\Pr(y_n \,|\, \lambda_n) = \lambda \exp\{-\lambda y_n\} \qquad \lambda_n = \exp\{\boldsymbol{w}^\mathsf{T} \boldsymbol{x}_n\}\,. \qquad (43)$$

Changelog

- 8 October 2018 – Initial version.