A note on maximum likelihood estimation

Sahand Negahban
S&DS 365/565

1 Introduction

The goal of this brief note is put down some foundations for maximum likelihood estimation. Throughout, we will assume that we observe some data Z and that we wish to estimate some parameters θ that govern the behavior of Z. There are a lot of ways to estimate such parameters. One way is through maximum likelihood estimation. To that end we simply write the likelihood for the parameter as

$$L(\theta) = \mathbb{P}_\theta(Z) \tag{1}$$

with the understanding that we use probability mass functions or probability density functions as necessary.

离散型：概率分布律 连续型：概率密度函数

判别

2 Regression problems from a discriminative perspective

In regression problems our data is $Z = \{(x_i, y_i)\}_{i=1}^n$. The fundamental assumption that we are making in the discriminative setting is that the conditional distribution of $p(y|x;\theta)$ is the only part of the data that depends on the underlying parameters.

One way to think about that is that the x_i are *fixed*, that is not random. At that point the only randomness comes from y_i and the distribution of y_i will depend on both x_i and the underlying parameter of interest θ. Often times, we might write equation (1) as

$$L(\theta) = \mathbb{P}(\{y_i\}_{i=1}^n; \{x_i\}_{i=1}^n, \theta)$$

where the use of the semi-colon means that the probability depends on the things that follow, but these are not random. They are fixed.

Other times, when we are talking about the values of x_i we won't do this and we will explicitly write down a conditional distribution model for y_i given x_i. At that point we will write down the conditional distribution and assume a distribution of the x_i. **However, we continue to assume that the conditional distribution of y given x is the only part that depends on the parameters.** Then, we can write down the following

$$L(\theta) = \mathbb{P}(\{y_i\}_{i=1}^n | \{x_i\}_{i=1}^n; \theta)\mathbb{P}(\{x_i\}_{i=1}^n)$$

Taking log we immediately see that the part that depends on θ is just the conditional distribution part and that we can ignore the distribution of x_i.

2.1 I.I.D. case

In the case that our data is generate **i.i.d.**, we can further break things up as

$$\begin{aligned} L(\theta) &= \prod_{i=1}^n \mathbb{P}(y_i, x_i; \theta) \\ &= \prod_{i=1}^n \mathbb{P}(y_i | x_i; \theta)\mathbb{P}(x_i) \end{aligned}$$

We again see that optimizing over θ is not influenced by the $\mathbb{P}(x_i)$ term. Thus, we will generally drop it from the likelihood and that gives us the conditional formulation that we are familiar with

$$L(\theta) = \prod_{i=1}^{n} \mathbb{P}(y_i | x_i; \theta)$$

Linear Regression via Maximization of the Likelihood

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

In least squares regression, we presented the common viewpoint that our approach to supervised learning be framed in terms of a loss function that scores our predictions relative to the ground truth as determined by the training data. That is, we introduced the idea of a function $\ell(\hat{y}, y)$ that is bigger when our machine learning model produces an estimate $\hat{y}$ that is worse relative to y. The loss function is a critical piece for turning the model-fitting problem into an optimization problem. In the case of least-squares regression, we used a squared loss:

$$\ell(\hat{y}, y) = (\hat{y} - y)^2 \,. \tag{1}$$

In this note we'll discuss a probabilistic view on constructing optimization problems that fit parameters to data by turning our loss function into a *likelihood*.

Maximizing the Likelihood

An alternative view on fitting a model is to think about a probabilistic procedure that might've given rise to the data. This probabilistic procedure would have parameters and then we can take the approach of trying to identify which parameters would assign the highest probability to the data that was observed. When we talk about probabilistic procedures that generate data given some parameters or covariates, we are really talking about *conditional probability distributions*.

Let's step away from regression for a minute and just talk about how we might think about a probabilistic procedure that generates data from a Gaussian distribution with a known variance but an unknown mean. Consider a set of data $\{y_n\}_{n=1}^N$ where $y_n \in \mathbb{R}$ and we assume that they are all independently and identically distributed according to a Gaussian distribution with unknown mean μ and variance σ^2. We would write this as a conditional probability as

$$y_n \,|\, \mu, \sigma^2 \sim \mathcal{N}(y_n \,|\, \mu, \sigma^2) \,. \tag{2}$$

The probability density function associate with this conditional distribution is the familiar univariate Gaussian:

$$\Pr(y_n \,|\, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ -\frac{1}{2\sigma^2}(y_n - \mu)^2 \right\} \,. \tag{3}$$

We have N i.i.d. data, however, and so we write the conditional distribution for all of them as a product:

$$\Pr(\{y_n\}_{n=1}^N \mid \mu, \sigma^2) = \prod_{n=1}^N \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y_n - \mu)^2\right\} . \tag{4}$$

This function, which we are here thinking of as *being parameterized by* μ is what we call the *likelihood*. When we maximize this with respect to μ, we are asking "what μ would assign the highest probability to the data we've seen?" This inductive criterion of selecting model parameters based on their ability to probabilistically explain the data is what we refer to as *maximum likelihood estimation* (MLE). Maximum likelihood estimation is a cornerstone of statistics and it has many wonderful properties that are out of scope for this course. At the end of the day, however, we can think of this as being a different (negative) loss function:

$$\mu^\star = \mu^{\mathsf{MLE}} = \arg\max_\mu \Pr(\{y_n\}_{n=1}^N \mid \mu, \sigma^2) = \arg\max_\mu \prod_{n=1}^N \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y_n - \mu)^2\right\} . \tag{5}$$

In practice, this isn't exactly the problem that we like to solve. Rather, we actually prefer to maximize the *log* likelihood because it turns all of our products into sums, which are easier to manipulate and differentiate, while preserving the location of the maximum. Also, when we take the product of many things that may be less than 1, the floating point numbers on our computer may become very close to zero and the maximization may not be numerically stable; taking the log makes those small positive numbers into better behaved negative numbers. As such, our (negative) loss function becomes

$$L(\mu) = \log \Pr(\{x_n\}_{n=1}^N \mid \mu, \sigma^2) = \sum_{n=1}^N \log\left(\frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y_n - \mu)^2\right\}\right) \tag{6}$$

$$= -N\log\sigma - \frac{N}{2}\log 2\pi - \frac{1}{2\sigma^2}\sum_{n=1}^N (y_n - \mu)^2 . \tag{7}$$

Figure 1 shows the likelihood function $L(\mu)$ that arises from a small set of data. Note in particular how the vertical scale of the likelihood is very small; this is one reason we transform it with the natural logarithm. We can go on and find the maximum likelihood estimate of μ by following the same kind of procedure that we used for least squares regression: differentiate, set to zero, and solve for μ:

$$\frac{d}{d\mu}L(\mu) = \frac{1}{\sigma^2}\sum_{n=1}^N (y_n - \mu) = 0 \tag{8}$$

$$\frac{1}{\sigma^2}\sum_{n=1}^N y_n - \frac{N}{\sigma^2}\mu = 0 \tag{9}$$

$$\mu = \frac{1}{N}\sum_{n=1}^N y_n . \tag{10}$$

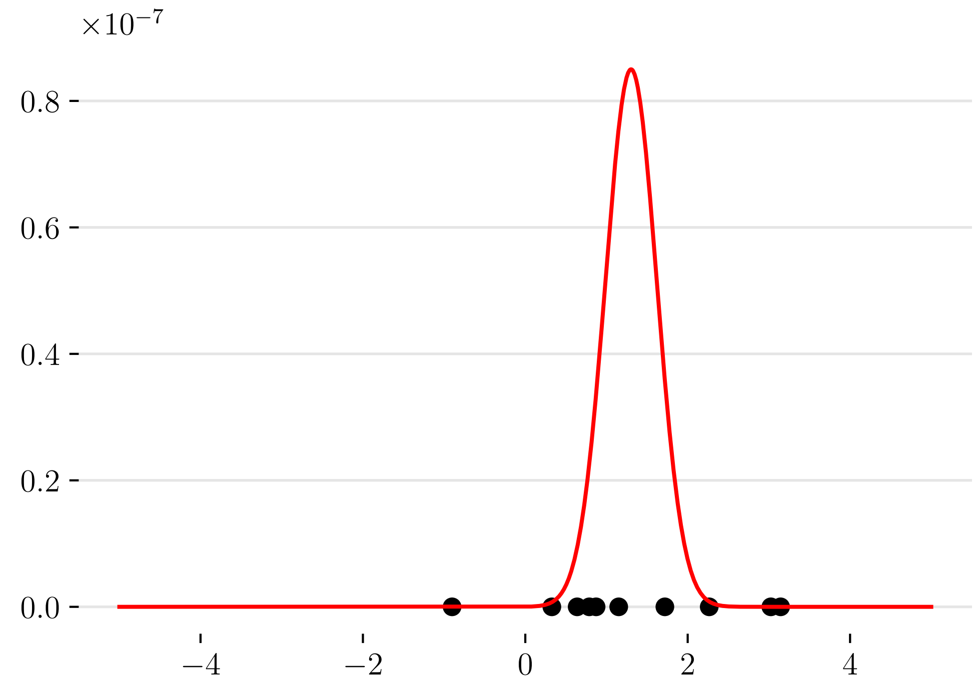

Figure 1: The black dots are ten ($N = 10$) data from a Gaussian distribution with $\sigma^2 = 1$ and $\mu = 1.4$. The red line is the likelihood as a function of μ. The maximum likelihood estimate is the peak of the red line. The red line is proportional to a Gaussian distribution but it is not generally true that likelihoods will have the same shape as the data distribution; this is a property that happens to arise when the location parameter is the quantity being estimated. Note also that the scale of the likelihood is tiny.

Unsurprisingly, the maximum likelihood estimate in this model (regardless of σ^2) is the sample average of the data.

MLE Regression with Gaussian Noise

We now revisit the linear regression problem with a maximum likelihood approach. As in the previous lecture, we assume our data are tuples of the form $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, where $\boldsymbol{x}_n \in \mathbb{R}^D$ and $y_n \in \mathbb{R}$. Rather than having a least-squares loss function, however, we now have to construct an explicit model for the noise that lets us reason about the conditional probability of the label. That is, we're now going to say that our data arise from a process like

$$y = \boldsymbol{x}^{\mathsf{T}}\boldsymbol{w} + \epsilon \tag{11}$$

where $\epsilon \in \mathbb{R}$ is a random variable capturing the noise. Just like we can construct different loss functions, we can think about different noise models for ϵ. Generalizing the previous section, a very natural idea is to say that this noise is from a zero-mean Gaussian distribution with variance σ^2, i.e.,

$$\epsilon \mid \sigma^2 \sim \mathcal{N}(\epsilon \mid 0, \sigma^2)\,. \tag{12}$$

Adding a constant to a Gaussian just has the effect of shifting its mean, so the resulting conditional probability distribution for our generative probabilistic process is

$$\Pr(y_n \mid \boldsymbol{x}_n, \boldsymbol{w}, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y_n - \boldsymbol{x}_n^{\mathsf{T}}\boldsymbol{w})^2\right\}\,. \tag{13}$$

This is very similar to Eq. 3, except now rather than conditioning on μ, we're conditioning on $\boldsymbol{x}_n$ and $\boldsymbol{w}$. Those two quantities combine to form the mean of the Gaussian distribution on y_n.

We denote the noise associated with the nth observation as ϵ_n and we will take these to be independent and identically distributed. This allows us to write the overall likelihood function as a product over these N terms:

$$\Pr(\{y_n\}_{n=1}^N \mid \{\boldsymbol{x}_n\}_{n=1}^N, \boldsymbol{w}, \sigma^2) = \prod_{n=1}^N \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y_n - \boldsymbol{x}_n^{\mathsf{T}}\boldsymbol{w})^2\right\}\,. \tag{14}$$

Again, this is a function of $\boldsymbol{w}$, as that is the parameter we are seeking to fit to the data.

In the previous lecture, we employed some more compact notation and aggregated the labels into a vector $\boldsymbol{y}$ and the features into a design matrix $\boldsymbol{X}$. We do the same trick here, but with a new twist: we're going to turn this univariate Gaussian distribution into a multivariate Gaussian distribution with a diagonal covariance matrix. Recall that the PDF for a D-dimensional Gaussian distribution is

$$\Pr(\boldsymbol{z} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = |\boldsymbol{\Sigma}|^{-1/2}(2\pi)^{-D/2} \exp\left\{-\frac{1}{2}(\boldsymbol{z} - \boldsymbol{\mu})^{\mathsf{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{z} - \boldsymbol{\mu})\right\}\,. \tag{15}$$

The covariance matrix $\boldsymbol{\Sigma}$ must be square, symmetric, and positive definite. When $\boldsymbol{\Sigma}$ is diagonal, the D dimensions are independent of each other. Putting our regression likelihood into this form we write:

$$\Pr(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma^2) = \mathcal{N}(\boldsymbol{y} \mid \boldsymbol{X}\boldsymbol{w}, \sigma^2\mathbf{I}) = (2\sigma^2\pi)^{-N/2} \exp\left\{-\frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\right\}\,. \tag{16}$$

We can now think about how we'd maximize this with respect to $\boldsymbol{w}$ in order to find the maximum likelihood estimate. As in the simple Gaussian case, it is helpful to take the natural log first:

$$\log \Pr(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \sigma^2) = -\frac{N}{2}\log(2\sigma^2\pi) - \frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\,. \tag{17}$$

The additive term doesn't have a $\boldsymbol{w}$. We are then left with the following optimization problem:

$$\boldsymbol{w}^{\mathsf{MLE}} = \arg\max_{\boldsymbol{w}} \left\{-\frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\right\}\,. \tag{18}$$

The $\frac{1}{2\sigma^2}$ does not change the solution to this problem and of course could change the sign and make this maximization into a minimization:

$$\boldsymbol{w}^{\mathsf{MLE}} = \arg\min_{\boldsymbol{w}} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})\,. \tag{19}$$

This is exactly the same optimization problem that we solved for the least-squares linear regression! While it seems like the loss function view and the maximum likelihood view are different, this reveals that they are often the same under the hood: least squares can be interpreted as assuming Gaussian noise, and particular choices of likelihood can be interpreted directly as (usually exponentiated) loss functions.

Fitting σ^2

One thing that is different about maximum likelihood, however, is that it gives us an additional parameter to play with that helps us reason about the *predictive distribution*. The predictive distribution is the distribution over the label, given parameters we have just fit. Rather than simply producing a single estimate, when we have a probabilistic model we can account for noise when we look at test data. That is, after finding $\boldsymbol{w}^{\mathsf{MLE}}$ if we have a query input $\boldsymbol{x}_{\mathsf{pred}}$ for which we don't know the $\boldsymbol{y}$, we could compute a guess via $y_{\mathsf{pred}} = \boldsymbol{x}_{\mathsf{pred}}^{\mathsf{T}}\boldsymbol{w}^{\mathsf{MLE}}$, or we could actually construct a whole distribution:

$$\Pr(y_{\mathsf{pred}} \,|\, \boldsymbol{x}_{\mathsf{pred}}, \boldsymbol{w}^{\mathsf{MLE}}, \sigma^2) = \mathcal{N}(y_{\mathsf{pred}} \,|\, \boldsymbol{x}_{\mathsf{pred}}^{\mathsf{T}}\boldsymbol{w}^{\mathsf{MLE}}, \sigma^2)\,. \tag{20}$$

This sounds great, but σ^2 went away when we constructed the optimization problem for $\boldsymbol{w}$. Why would it be any good? Well, it won't be any good — unless you fit it also. Fortunately, maximum likelihood estimation tells us how to do that one also, and we can start out by assuming that we've already computed $\boldsymbol{w}^{\mathsf{MLE}}$. We set up the problem the same way except we keep the additive term in Eq. 17:

$$\sigma^{\mathsf{MLE}} = \arg\max_{\sigma} \left\{ -\frac{N}{2}\log(2\sigma^2\pi) - \frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y}) \right\}\,. \tag{21}$$

Solving this maximization problem is again just a question of differentiating and setting to zero:

$$\frac{\partial}{\partial\sigma^2}\left[-\frac{N}{2}\log\sigma^2 - \frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y}) \right] = 0 \tag{22}$$

$$-\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y}) = 0 \tag{23}$$

$$-N + \frac{1}{\sigma^2}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y}) = 0 \tag{24}$$

$$\sigma^2 = \frac{1}{N}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})^{\mathsf{T}}(\boldsymbol{X}\boldsymbol{w}^{\mathsf{MLE}} - \boldsymbol{y})\,. \tag{25}$$

This is a satisfying result because it is just finding the sample average of the squared deviations between what $\boldsymbol{w}^{\mathsf{MLE}}$ predicts and what the training data actually are. It feels exactly like what happens when you compute the maximum likelihood estimate of the variance of a univariate Gaussian distribution.

Changelog

- 17 September 2018 – Initial version

Computing Neural Network Gradients

Kevin Clark

1 Introduction

The purpose of these notes is to demonstrate how to quickly compute neural network gradients in a completely vectorized way. It is complementary to the last part of lecture 3 in CS224n 2019, which goes over the same material.

2 Vectorized Gradients

While it is a good exercise to compute the gradient of a neural network with respect to a single parameter (e.g., a single element in a weight matrix), in practice this tends to be quite slow. Instead, it is more efficient to keep everything in matrix/vector form. The basic building block of vectorized gradients is the *Jacobian Matrix*. Suppose we have a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ that maps a vector of length n to a vector of length m: $\boldsymbol{f}(\boldsymbol{x}) = [f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n)]$. Then its Jacobian is the following $m \times n$ matrix:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

That is, $(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial f_i}{\partial x_j}$ (which is just a standard non-vector derivative). The Jacobian matrix will be useful for us because we can apply the chain rule to a vector-valued function just by multiplying Jacobians.

As a little illustration of this, suppose we have a function $\boldsymbol{f}(x) = [f_1(x), f_2(x)]$ taking a scalar to a vector of size 2 and a function $\boldsymbol{g}(\boldsymbol{y}) = [g_1(y_1, y_2), g_2(y_1, y_2)]$ taking a vector of size two to a vector of size two. Now let's compose them to get $\boldsymbol{g}(x) = [g_1(f_1(x), f_2(x)), g_2(f_1(x), f_2(x))]$. Using the regular chain rule, we can compute the derivative of $\boldsymbol{g}$ as the Jacobian

$$\frac{\partial \boldsymbol{g}}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x} g_1(f_1(x), f_2(x)) \\ \frac{\partial}{\partial x} g_2(f_1(x), f_2(x)) \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1}\frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2}\frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1}\frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2}\frac{\partial f_2}{\partial x} \end{bmatrix}$$

And we see this is the same as multiplying the two Jacobians:

$$\frac{\partial \boldsymbol{g}}{\partial x} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{bmatrix}$$

3 Useful Identities

This section will now go over how to compute the Jacobian for several simple functions. It will provide some useful identities you can apply when taking neural network gradients.

(1) **Matrix times column vector with respect to the column vector** ($\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)

Suppose $\boldsymbol{W} \in \mathbb{R}^{n\times m}$. Then we can think of $\boldsymbol{z}$ as a function of $\boldsymbol{x}$ taking an m-dimensional vector to an n-dimensional vector. So its Jacobian will be $n \times m$. Note that

$$z_i = \sum_{k=1}^{m} W_{ik}x_k$$

So an entry $(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij}$ of the Jacobian will be

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j}\sum_{k=1}^{m} W_{ik}x_k = \sum_{k=1}^{m} W_{ik}\frac{\partial}{\partial x_j}x_k = W_{ij}$$

because $\frac{\partial}{\partial x_j}x_k = 1$ if $k = j$ and 0 if otherwise. So we see that $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W}}$

(2) **Row vector times matrix with respect to the row vector** ($\boldsymbol{z} = \boldsymbol{x}\boldsymbol{W}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)

A computation similar to (1) shows that $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W}^T}$.

(3) **A vector with itself**
($\boldsymbol{z} = \boldsymbol{x}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)
We have $z_i = x_i$. So

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j}x_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

So we see that the Jacobian $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a diagonal matrix where the entry at (i, i) is 1. This is just the identity matrix: $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{I}}$. When applying the chain rule, this term will disappear because a matrix or vector multiplied by the identity matrix does not change.

(4) **An elementwise function applied a vector**
($\boldsymbol{z} = f(\boldsymbol{x})$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)
Since f is being applied elementwise, we have $z_i = f(x_i)$. So

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} f(x_i) = \begin{cases} f'(x_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

So we see that the Jacobian $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a diagonal matrix where the entry at (i, i) is the derivative of f applied to x_i. We can write this as $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \text{diag}(f'(\boldsymbol{x}))}$. Since multiplication by a diagonal matrix is the same as doing elementwise multiplication by the diagonal, we could also write $\boxed{\circ f'(\boldsymbol{x})}$ when applying the chain rule.

(5) **Matrix times column vector with respect to the matrix**
($\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, $\boldsymbol{\delta} = \frac{\partial J}{\partial \boldsymbol{z}}$ what is $\frac{\partial J}{\partial \boldsymbol{W}} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$?)

This is a bit more complicated than the other identities. The reason for including $\frac{\partial J}{\partial \boldsymbol{z}}$in the above problem formulation will become clear in a moment.

First suppose we have a loss function J (a scalar) and are computing its gradient with respect to a matrix $\boldsymbol{W} \in \mathbb{R}^{n \times m}$. Then we could think of J as a function of $\boldsymbol{W}$ taking nm inputs (the entries of $\boldsymbol{W}$) to a single output (J). This means the Jacobian $\frac{\partial J}{\partial \boldsymbol{W}}$ would be a $1 \times nm$ vector. But in practice this is not a very useful way of arranging the gradient. It would be much nicer if the derivatives were in a $n \times m$ matrix like this:

$$\frac{\partial J}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \cdots & \frac{\partial J}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{n1}} & \cdots & \frac{\partial J}{\partial W_{nm}} \end{bmatrix}$$

Since this matrix has the same shape as $\boldsymbol{W}$, we could just subtract it (times the learning rate) from $\boldsymbol{W}$ when doing gradient descent. So (in a slight abuse of notation) let's find this matrix as $\frac{\partial J}{\partial \boldsymbol{W}}$ instead.

This way of arranging the gradients becomes complicated when computing $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$. Unlike J, $\boldsymbol{z}$ is a vector. So if we are trying to rearrange the gradients like with $\frac{\partial J}{\partial \boldsymbol{W}}$, $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ would be an $n \times m \times n$ tensor! Luckily, we can avoid the issue by taking the gradient with respect to a single weight W_{ij} instead.

$\frac{\partial \boldsymbol{z}}{\partial W_{ij}}$ is just a vector, which is much easier to deal with. We have

$$z_k = \sum_{l=1}^{m} W_{kl} x_l$$
$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{m} x_l \frac{\partial}{\partial W_{ij}} W_{kl}$$

Note that $\frac{\partial}{\partial W_{ij}} W_{kl} = 1$ if $i = k$ and $j = l$ and 0 if otherwise. So if $k \neq i$ everything in the sum is zero and the gradient is zero. Otherwise, the only nonzero element of the sum is when $l = j$, so we just get x_j. Thus we find $\frac{\partial z_k}{\partial W_{ij}} = x_j$ if $k = i$ and 0 if otherwise. Another way of writing this is

$$\frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i\text{th element}$$

Now let's compute $\frac{\partial J}{\partial W_{ij}}$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \sum_{k=1}^{m} \delta_k \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j$$

(the only nonzero term in the sum is $\delta_i \frac{\partial z_i}{\partial W_{ij}}$). To get $\frac{\partial J}{\partial \boldsymbol{W}}$ we want a matrix where entry (i, j) is $\delta_i x_j$. This matrix is equal to the outer product $\boxed{\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \boldsymbol{x}^T}$

(6) **Row vector time matrix with respect to the matrix** ($\boldsymbol{z} = \boldsymbol{x}\boldsymbol{W}$, $\boldsymbol{\delta} = \frac{\partial J}{\partial \boldsymbol{z}}$ what is $\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$?)

A similar computation to (5) shows that $\boxed{\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{x}^T \boldsymbol{\delta}}$.

(7) **Cross-entropy loss with respect to logits** ($\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{\theta})$, $J = CE(\boldsymbol{y}, \hat{\boldsymbol{y}})$, what is $\frac{\partial J}{\partial \boldsymbol{\theta}}$?)

The gradient is $\boxed{\frac{\partial J}{\partial \boldsymbol{\theta}} = \hat{\boldsymbol{y}} - \boldsymbol{y}}$
(or $(\hat{\boldsymbol{y}} - \boldsymbol{y})^T$ if $\boldsymbol{y}$ is a column vector).

These identities will be enough to let you quickly compute the gradients for many neural networks. However, it's important to know how to compute Jacobians for other functions as well in case they show up. Some examples if you want practice: dot product of two vectors, elementwise product of two vectors, 2-norm of a vector. Feel free to use these identities in the assignments. One option is just to memorize them. Another option is to figure them out by looking at the dimensions. For example, only one ordering/orientation of $\boldsymbol{\delta}$ and $\boldsymbol{x}$ will produce the correct shape for $\frac{\partial J}{\partial \boldsymbol{W}}$ (assuming $\boldsymbol{W}$ is not square).

4 Gradient Layout

Jacobean formulation is great for applying the chain rule: you just have to multiply the Jacobians. However, when doing SGD it's more convenient to follow the convention "the shape of the gradient equals the shape of the parameter" (as we did when computing $\frac{\partial J}{\partial \boldsymbol{W}}$). That way subtracting the gradient times the learning rate from the parameters is easy. **We expect answers to homework questions to follow this convention.** Therefore if you compute the gradient of a column vector using Jacobian formulation, you should take the transpose when reporting your final answer so the gradient is a column vector. Another option is to always follow the convention. In this case the identities may not work, but you can still figure out the answer by making sure the dimensions of your derivatives match up. Up to you which of these options you choose!

5 Example: 1-Layer Neural Network

This section provides an example of computing the gradients of a full neural network. In particular we are going to compute the gradients of a one-layer neural network trained with cross-entropy loss. The forward pass of the model is as follows:

$$\begin{aligned}
\boldsymbol{x} &= \text{input} \\
\boldsymbol{z} &= \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}_1 \\
\boldsymbol{h} &= \text{ReLU}(\boldsymbol{z}) \\
\boldsymbol{\theta} &= \boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2 \\
\hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{\theta}) \\
J &= CE(\boldsymbol{y}, \hat{\boldsymbol{y}})
\end{aligned}$$

It helps to break up the model into the simplest parts possible, so note that we defined $\boldsymbol{z}$ and $\boldsymbol{\theta}$ to split up the activation functions from the linear transformations in the network's layers. The dimensions of the model's parameters are

$$\boldsymbol{x} \in \mathbb{R}^{D_x \times 1} \qquad \boldsymbol{b}_1 \in \mathbb{R}^{D_h \times 1} \qquad \boldsymbol{W} \in \mathbb{R}^{D_h \times D_x} \qquad \boldsymbol{b}_2 \in \mathbb{R}^{N_c \times 1} \qquad \boldsymbol{U} \in \mathbb{R}^{N_c \times D_h}$$

where D_x is the size of our input, D_h is the size of our hidden layer, and N_c is the number of classes.

In this example, we will compute all of the network's gradients:

$$\frac{\partial J}{\partial \boldsymbol{U}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_2} \qquad \frac{\partial J}{\partial \boldsymbol{W}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_1} \qquad \frac{\partial J}{\partial \boldsymbol{x}}$$

To start with, recall that $\text{ReLU}(x) = \max(x, 0)$. This means

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if otherwise} \end{cases} = \text{sgn}(\text{ReLU}(x))$$

where sgn is the signum function. Note that we are able to write the derivative of the activation in terms of the activation itself.

Now let's write out the chain rule for $\frac{\partial J}{\partial \boldsymbol{U}}$ and $\frac{\partial J}{\partial \boldsymbol{b}_2}$:

$$\frac{\partial J}{\partial \boldsymbol{U}} = \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}}$$
$$\frac{\partial J}{\partial \boldsymbol{b}_2} = \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2}$$

Notice that $\frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} = \frac{\partial J}{\partial \boldsymbol{\theta}}$ is present in both gradients. This makes the math a bit cumbersome. Even worse, if we're implementing the model without automatic differentiation, computing $\frac{\partial J}{\partial \boldsymbol{\theta}}$ twice will be inefficient. So it will help us to define some variables to represent the intermediate derivatives:

$$\boldsymbol{\delta}_1 = \frac{\partial J}{\partial \boldsymbol{\theta}} \qquad \boldsymbol{\delta}_2 = \frac{\partial J}{\partial \boldsymbol{z}}$$

These can be thought as the error signals passed down to $\boldsymbol{\theta}$ and $\boldsymbol{z}$ when doing backpropagation. We can compute them as follows:

$$\begin{aligned}
\boldsymbol{\delta}_1 &= \frac{\partial J}{\partial \boldsymbol{\theta}} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T && \text{this is just identity (7)} \\
\boldsymbol{\delta}_2 &= \frac{\partial J}{\partial \boldsymbol{z}} = \frac{\partial J}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{using the chain rule} \\
&= \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{substituting in } \boldsymbol{\delta}_1 \\
&= \boldsymbol{\delta}_1 \; \boldsymbol{U} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{using identity (1)} \\
&= \boldsymbol{\delta}_1 \; \boldsymbol{U} \circ \text{ReLU}'(\boldsymbol{z}) && \text{using identity (4)} \\
&= \boldsymbol{\delta}_1 \; \boldsymbol{U} \circ \text{sgn}(\boldsymbol{h}) && \text{we computed this earlier}
\end{aligned}$$

A good way of checking our work is by looking at the dimensions of the Jacobians:

$$\begin{array}{ccccccc}
\frac{\partial J}{\partial \boldsymbol{z}} & = & \boldsymbol{\delta}_1 & & \boldsymbol{U} & \circ & \text{sgn}(\boldsymbol{h}) \\
(1 \times D_h) & & (1 \times N_c) & & (N_c \times D_h) & & (D_h)
\end{array}$$

We see that the dimensions of all the terms in the gradient match up (i.e., the number of columns in a term equals the number of rows in the next term). This will always be the case if we computed our gradients correctly.

Now we can use the error terms to compute our gradients. Note that we transpose out answers when computing the gradients for column vectors terms to follow the shape convention.

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{U}} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}} = \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}} = \boldsymbol{\delta}_1^T \boldsymbol{h}^T && \text{using identity (5)} \\
\frac{\partial J}{\partial \boldsymbol{b}_2} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2} = \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2} = \boldsymbol{\delta}_1^T && \text{using identity (3) and transposing} \\
\frac{\partial J}{\partial \boldsymbol{W}} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}_2 \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}_2^T \boldsymbol{x}^T && \text{using identity (5)} \\
\frac{\partial J}{\partial \boldsymbol{b}_1} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_2 \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_2^T && \text{using identity (3) and transposing} \\
\frac{\partial J}{\partial \boldsymbol{x}} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = (\boldsymbol{\delta}_2 \boldsymbol{W})^T && \text{using identity (1) and transposing}
\end{aligned}$$

Chris Piech
CS109

Handout #40
May 20th, 2016

Logistic Regression

Before we get started I wanted to familiarize you with some notation:

$$\theta^T \mathbf{x} = \sum_{i=1}^{n} \theta_i x_i = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \qquad \text{weighted sum}$$

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{sigmoid function}$$

Logistic Regression Overview

Classification is the task of choosing a value of y that maximizes $P(Y|X)$. Naïve Bayes worked by approximating that probability using the naïve assumption that each feature was independent given the class label.

For all classification algorithms you are given n I.I.D. training datapoints $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots (\mathbf{x}^{(n)}, y^{(n)})$ where each "feature" vector $\mathbf{x}^{(i)}$ has $m = |\mathbf{x}^{(i)}|$ features.

Logistic Regression Assumption

Logistic Regression is a classification algorithm (I know, terrible name) that works by trying to learn a function that approximates $P(Y|X)$. It makes the central assumption that $P(Y|X)$ can be approximated as a sigmoid function applied to a linear combination of input features. Mathematically, for a single training datapoint $(\mathbf{x}, y)$ Logistic Regression assumes:

$$P(Y=1|\mathbf{X}=\mathbf{x}) = \sigma(z) \text{ where } z = \theta_0 + \sum_{i=1}^{m} \theta_i x_i$$

This assumption is often written in the equivalent forms:

$$P(Y=1|\mathbf{X}=\mathbf{x}) = \sigma(\theta^T \mathbf{x}) \qquad \text{where we always set } x_0 \text{ to be 1}$$

$$P(Y=0|\mathbf{X}=\mathbf{x}) = 1 - \sigma(\theta^T \mathbf{x}) \qquad \text{by total law of probability}$$

Using these equations for probability of $Y|X$ we can create an algorithm that select values of theta that maximize that probability for all data. I am first going to state the log probability function and partial derivatives with respect to theta. Then later we will (a) show an algorithm that can chose optimal values of theta and (b) show how the equations were derived.

Log Likelihood

We can write an equation for the likelihood of all the data (under the Logistic Regression assumption). If you take the log of the likelihood equation the result is:

$$LL(\theta) = \sum_{i=0}^{n} y^{(i)} \log \sigma(\theta^T \mathbf{x}^{(i)}) + (1 - y^{(i)}) \log[1 - \sigma(\theta^T \mathbf{x}^{(i)})]$$

We will show the derivation later.

Gradient of Log Likelihood

Now that we have a function for log-likelihood, we simply need to chose the values of theta that maximize it. Unlike it other questions, there is no closed form way to calculate theta. Instead we chose it using optimization. Here is the partial derivative of log-likelihood with respect to each parameter θ_j:

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \sum_{i=0}^{n} \left[y^{(i)} - \sigma(\theta^T \mathbf{x}^{(i)}) \right] x_j^{(i)}$$

Gradient Ascent Optimization

Once we have an equation for Log Likelihood, we chose the values for our parameters (θ) that maximize said function. In the case of logistic regression we can't solve for θ mathematically. Instead we use a computer to chose θ. To do so we employ an algorithm called gradient ascent. That algorithms claims that if you continuously take small steps in the direction of your gradient, you will eventually make it to a local maxima. In the case of Logistic Regression you can prove that the result will always be a global maxima.

The small step that we continually take given the training dataset can be calculated as:

$$\begin{aligned}\theta_j^{\text{new}} &= \theta_j^{\text{old}} + \eta \cdot \frac{\partial LL(\theta^{\text{ old}})}{\partial \theta_j^{\text{old}}} \\ &= \theta_j^{\text{old}} + \eta \cdot \sum_{i=0}^{n} \left[y^{(i)} - \sigma(\theta^T \mathbf{x}^{(i)}) \right] x_j^{(i)}\end{aligned}$$

Where η is the magnitude of the step size that we take. If you keep updating θ using the equation above you will converge on the best values of θ!

Derivations

In this section we provide the mathematical derivations for the log-likelihood function and the gradient. The derivations are worth knowing because these ideas are heavily used in Neural Networks. To start, here is a super slick way of writing the probability of one datapoint:

$$P(Y = y|X = \mathbf{x}) = \sigma(\theta^T \mathbf{x})^y \cdot \left[1 - \sigma(\theta^T \mathbf{x})\right]^{(1-y)}$$

Since each datapoint is independent, the probability of all the data is:

$$\begin{aligned}L(\theta) &= \prod_{i=1}^{n} P(Y = y^{(i)}|X = \mathbf{x}^{(i)}) \\ &= \prod_{i=1}^{n} \sigma(\theta^T \mathbf{x}^{(i)})^{y^{(i)}} \cdot \left[1 - \sigma(\theta^T \mathbf{x}^{(i)})\right]^{(1-y^{(i)})}\end{aligned}$$

And if you take the log of this function, you get the reported Log Likelihood for Logistic Regression.

The next step is to calculate the derivative of the log likelihood with respect to each theta. To start, here is the definition for the derivative of sigma with respect to its inputs:

$$\frac{\partial}{\partial z}\sigma(z) = \sigma(z)[1 - \sigma(z)] \qquad \text{to get the derivative with respect to } \theta\text{, use the chain rule}$$

Derivative of gradient for one datapoint $(\mathbf{x}, y)$:

$$\begin{aligned}\frac{\partial LL(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} y \log \sigma(\theta^T \mathbf{x}) + \frac{\partial}{\partial \theta_j}(1-y)\log[1 - \sigma(\theta^T \mathbf{x}] && \text{derivative of sum of terms} \\ &= \left[\frac{y}{\sigma(\theta^T x)} - \frac{1-y}{1-\sigma(\theta^T x)}\right] \frac{\partial}{\partial \theta_j}\sigma(\theta^T x) && \text{derivative of } \log f(x) \\ &= \left[\frac{y}{\sigma(\theta^T x)} - \frac{1-y}{1-\sigma(\theta^T x)}\right] \sigma(\theta^T x)[1-\sigma(\theta^T x)]x_j && \text{chain rule + derivative of sigma} \\ &= \left[\frac{y - \sigma(\theta^T x)}{\sigma(\theta^T x)[1-\sigma(\theta^T x)]}\right] \sigma(\theta^T x)[1-\sigma(\theta^T x)]x_j && \text{algebraic manipulation} \\ &= \left[y - \sigma(\theta^T x)\right] x_j && \text{cancelling terms}\end{aligned}$$

Because the derivative of sums is the sum of derivatives, the gradient of theta is simply the sum of this term for each training datapoint.