Chapter 2

Classification and logistic regression

Let's now talk about the classification problem. This is just like the regression problem, except that the values y we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification** problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. 0 is also called the **negative class**, and 1 the **positive class**, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example.

2.1 Logistic regression

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$.

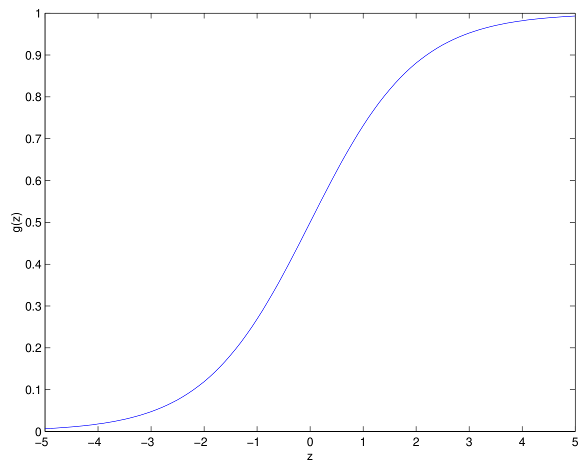
To fix this, let's change the form for our hypotheses $h_\theta(x)$. We will choose

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

where

$$g(z) = \frac{1}{1 + e^{-z}}$$

is called the **logistic function** or the **sigmoid function**. Here is a plot showing $g(z)$:



Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^d \theta_j x_j$.

For now, let's take the choice of g as given. Other functions that smoothly increase from 0 to 1 can also be used, but for a couple of reasons that we'll see later (when we talk about GLMs, and when we talk about generative learning algorithms), the choice of the logistic function is a fairly natural one. Before moving on, here's a useful property of the derivative of the sigmoid function, which we write as g':

$$\begin{aligned} g'(z) &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2} \left(e^{-z}\right) \\ &= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\ &= g(z)(1-g(z)). \end{aligned}$$

So, given the logistic regression model, how do we fit θ for it? Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, let's endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood.

Let us assume that

$$\begin{aligned} P(y = 1 \mid x; \theta) &= h_\theta(x) \\ P(y = 0 \mid x; \theta) &= 1 - h_\theta(x) \end{aligned}$$

Note that this can be written more compactly as

$$p(y \mid x; \theta) = (h_\theta(x))^y \left(1 - h_\theta(x)\right)^{1-y}$$

Assuming that the n training examples were generated independently, we can then write down the likelihood of the parameters as

$$\begin{aligned} L(\theta) &= p(\vec{y} \mid X; \theta) \\ &= \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^{n} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}} \end{aligned}$$

As before, it will be easier to maximize the log likelihood:

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{n} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \tag{2.1}$$

How do we maximize the likelihood? Similar to our derivation in the case of linear regression, we can use gradient ascent. Written in vectorial notation, our updates will therefore be given by $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$. (Note the positive rather than negative sign in the update formula, since we're maximizing, rather than minimizing, a function now.) Let's start by working with just one training example (x, y), and take derivatives to derive the stochastic gradient ascent rule:

$$\begin{aligned} \frac{\partial}{\partial \theta_j} \ell(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\ &= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) g(\theta^T x)(1 - g(\theta^T x)) \frac{\partial}{\partial \theta_j} \theta^T x \\ &= \left(y(1 - g(\theta^T x)) - (1 - y) g(\theta^T x) \right) x_j \\ &= \left(y - h_\theta(x) \right) x_j \end{aligned} \tag{2.2}$$

Above, we used the fact that $g'(z) = g(z)(1 - g(z))$. This therefore gives us the stochastic gradient ascent rule

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$$

If we compare this to the LMS update rule, we see that it looks identical; but this is *not* the same algorithm, because $h_\theta(x^{(i)})$ is now defined as a non-linear function of $\theta^T x^{(i)}$. Nonetheless, it's a little surprising that we end up with the same update rule for a rather different algorithm and learning problem. Is this coincidence, or is there a deeper reason behind this? We'll answer this when we get to GLM models.

Remark 2.1.1: An alternative notational viewpoint of the same loss function is also useful, especially for Section 7.1 where we study nonlinear models. Let $\ell_{\text{logistic}} : \mathbb{R} \times \{0, 1\} \to \mathbb{R}_{\geq 0}$ be the *logistic loss* defined as

$$\ell_{\text{logistic}}(t, y) \triangleq y \log(1 + \exp(-t)) + (1 - y) \log(1 + \exp(t)) \,. \tag{2.3}$$

One can verify by plugging in $h_\theta(x) = 1/(1 + e^{-\theta^\top x})$ that the *negative* log-likelihood (the negation of $\ell(\theta)$ in equation (2.1)) can be re-written as

$$-\ell(\theta) = \ell_{\text{logistic}}(\theta^\top x, y). \tag{2.4}$$

Oftentimes $\theta^\top x$ or t is called the *logit*. Basic calculus gives us that

$$\begin{aligned}
\frac{\partial \ell_{\text{logistic}}(t, y)}{\partial t} &= y \frac{-\exp(-t)}{1 + \exp(-t)} + (1 - y) \frac{1}{1 + \exp(-t)} && (2.5) \\
&= 1/(1 + \exp(-t)) - y. && (2.6)
\end{aligned}$$

Then, using the chain rule, we have that

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} \ell(\theta) &= -\frac{\partial \ell_{\text{logistic}}(t, y)}{\partial t} \cdot \frac{\partial t}{\partial \theta_j} && (2.7) \\
&= (y - 1/(1 + \exp(-t))) \cdot x_j = (y - h_\theta(x)) x_j \,, && (2.8)
\end{aligned}$$

which is consistent with the derivation in equation (2.2). We will see this viewpoint can be extended nonlinear models in Section 7.1.

2.2 Digression: the perceptron learning algorithm

We now digress to talk briefly about an algorithm that's of some historical interest, and that we will also return to later when we talk about learning theory. Consider modifying the logistic regression method to "force" it to output values that are either 0 or 1 or exactly. To do so, it seems natural to change the definition of g to be the threshold function:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If we then let $h_\theta(x) = g(\theta^T x)$ as before but using this modified definition of g, and if we use the update rule

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

then we have the **perceptron learning algorithn**.

In the 1960s, this "perceptron" was argued to be a rough model for how individual neurons in the brain work. Given how simple the algorithm is, it will also provide a starting point for our analysis when we talk about learning theory later in this class. Note however that even though the perceptron may be cosmetically similar to the other algorithms we talked about, it is actually a very different type of algorithm than logistic regression and least squares linear regression; in particular, it is difficult to endow the perceptron's predictions with meaningful probabilistic interpretations, or derive the perceptron as a maximum likelihood estimation algorithm.

2.3 Multi-class classification

Consider a classification problem in which the response variable y can take on any one of k values, so $y \in \{1, 2, \ldots, k\}$. For example, rather than classifying emails into the two classes spam or not-spam—which would have been a binary classification problem—we might want to classify them into three classes, such as spam, personal mails, and work-related mails. The label / response variable is still discrete, but can now take on more than two values. We will thus model it as distributed according to a multinomial distribution.

In this case, $p(y \mid x; \theta)$ is a distribution over k possible discrete outcomes and is thus a multinomial distribution. Recall that a multinomial distribution involves k numbers $\phi_1, \ldots, \phi_k$ specifying the probability of each of the outcomes. Note that these numbers must satisfy $\sum_{i=1}^k \phi_i = 1$. We will design a parameterized model that outputs $\phi_1, \ldots, \phi_k$ satisfying this constraint given the input x.

We introduce k groups of parameters $\theta_1, \ldots, \theta_k$, each of them being a vector in $\mathbb{R}^d$. Intuitively, we would like to use $\theta_1^\top x, \ldots, \theta_k^\top x$ to represent

$\phi_1, \dots, \phi_k$, the probabilities $P(y = 1 \mid x; \theta), \dots, P(y = k \mid x; \theta)$. However, there are two issues with such a direct approach. First, $\theta_j^\top x$ is not necessarily within $[0, 1]$. Second, the summation of $\theta_j^\top x$'s is not necessarily 1. Thus, instead, we will use the softmax function to turn $(\theta_1^\top x, \cdots, \theta_k^\top x)$ into a probability vector with nonnegative entries that sum up to 1.

Define the softmax function softmax : $\mathbb{R}^k \to \mathbb{R}^k$ as

$$\text{softmax}(t_1, \dots, t_k) = \begin{bmatrix} \frac{\exp(t_1)}{\sum_{j=1}^k \exp(t_j)} \\ \vdots \\ \frac{\exp(t_k)}{\sum_{j=1}^k \exp(t_j)} \end{bmatrix}. \tag{2.9}$$

The inputs to the softmax function, the vector t here, are often called *logits*. Note that by definition, the output of the softmax function is always a probability vector whose entries are nonnegative and sum up to 1.

Let $(t_1, \dots, t_k) = (\theta_1^\top x, \cdots, \theta_k^\top x)$. We apply the softmax function to $(t_1, \dots, t_k)$, and use the output as the probabilities $P(y = 1 \mid x; \theta), \dots, P(y = k \mid x; \theta)$. We obtain the following probabilistic model:

$$\begin{bmatrix} P(y = 1 \mid x; \theta) \\ \vdots \\ P(y = k \mid x; \theta) \end{bmatrix} = \text{softmax}(t_1, \cdots, t_k) = \begin{bmatrix} \frac{\exp(\theta_1^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)} \\ \vdots \\ \frac{\exp(\theta_k^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)} \end{bmatrix}. \tag{2.10}$$

For notational convenience, we will let $\phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)}$. More succinctly, the equation above can be written as:

$$P(y = i \mid x; \theta) = \phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)} = \frac{\exp(\theta_i^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)}. \tag{2.11}$$

Next, we compute the negative log-likelihood of a single example (x, y).

$$-\log p(y \mid x, \theta) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^k \exp(t_j)}\right) = -\log\left(\frac{\exp(\theta_y^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)}\right) \tag{2.12}$$

Thus, the loss function, the negative log-likelihood of the training data, is given as

$$\ell(\theta) = \sum_{i=1}^n -\log\left(\frac{\exp(\theta_{y^{(i)}}^\top x^{(i)})}{\sum_{j=1}^k \exp(\theta_j^\top x^{(i)})}\right). \tag{2.13}$$

It's convenient to define the cross-entropy loss $\ell_{\text{ce}} : \mathbb{R}^k \times \{1, \ldots, k\} \to \mathbb{R}_{\geq 0}$, which modularizes in the complex equation above:[1]

$$\ell_{\text{ce}}((t_1, \ldots, t_k), y) = -\log\left(\frac{\exp(t_y)}{\sum_{j=1}^k \exp(t_j)}\right). \tag{2.14}$$

With this notation, we can simply rewrite equation (2.13) as

$$\ell(\theta) = \sum_{i=1}^n \ell_{\text{ce}}((\theta_1^\top x^{(i)}, \ldots, \theta_k^\top x^{(i)}), y^{(i)}). \tag{2.15}$$

Moreover, conveniently, the cross-entropy loss also has a simple gradient. Let $t = (t_1, \ldots, t_k)$, and recall $\phi_i = \frac{\exp(t_i)}{\sum_{j=1}^k \exp(t_j)}$. By basic calculus, we can derive

$$\frac{\partial \ell_{\text{ce}}(t, y)}{\partial t_i} = \phi_i - 1\{y = i\}, \tag{2.16}$$

where $1\{\cdot\}$ is the indicator function, that is, $1\{y = i\} = 1$ if $y = i$, and $1\{y = i\} = 0$ if $y \neq i$. Alternatively, in vectorized notations, we have the following form which will be useful for Chapter 7:

$$\frac{\partial \ell_{\text{ce}}(t, y)}{\partial t} = \phi - e_y, \tag{2.17}$$

where $e_s \in \mathbb{R}^k$ is the s-th natural basis vector (where the s-th entry is 1 and all other entries are zeros.) Using Chain rule, we have that

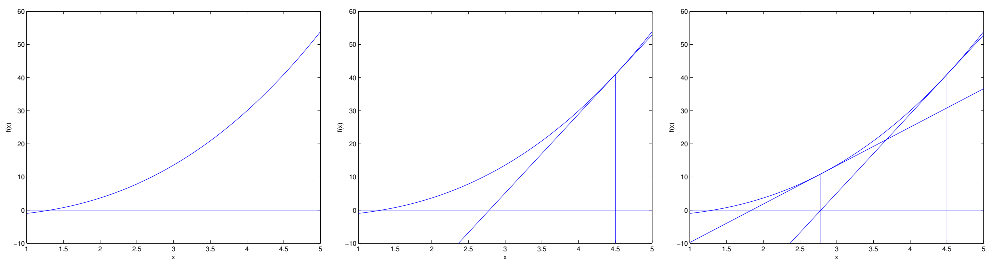
$$\frac{\partial \ell_{\text{ce}}((\theta_1^\top x, \ldots, \theta_k^\top x), y)}{\partial \theta_i} = \frac{\partial \ell(t, y)}{\partial t_i} \cdot \frac{\partial t_i}{\partial \theta_i} = (\phi_i - 1\{y = i\}) \cdot x. \tag{2.18}$$

Therefore, the gradient of the loss with respect to the part of parameter θ_i is

$$\frac{\partial \ell(\theta)}{\partial \theta_i} = \sum_{j=1}^n (\phi_i^{(j)} - 1\{y^{(j)} = i\}) \cdot x^{(j)}, \tag{2.19}$$

where $\phi_i^{(j)} = \frac{\exp(\theta_i^\top x^{(j)})}{\sum_{s=1}^k \exp(\theta_s^\top x^{(j)})}$ is the probability that the model predicts item i for example $x^{(j)}$. With the gradients above, one can implement (stochastic) gradient descent to minimize the loss function $\ell(\theta)$.

[1]There are some ambiguity in the naming here. Some people call the cross-entropy loss the function that maps the probability vector (the ϕ in our language) and label y to the final real number, and call our version of cross-entropy loss softmax-cross-entropy loss. We choose our current naming convention because it's consistent with the naming of most modern deep learning library such as PyTorch and Jax.



2.4 Another algorithm for maximizing $\ell(\theta)$

Returning to logistic regression with $g(z)$ being the sigmoid function, let's now talk about a different algorithm for maximizing $\ell(\theta)$.

To get us started, let's consider Newton's method for finding a zero of a function. Specifically, suppose we have some function $f : \mathbb{R} \mapsto \mathbb{R}$, and we wish to find a value of θ so that $f(\theta) = 0$. Here, $\theta \in \mathbb{R}$ is a real number. Newton's method performs the following update:

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)}.$$

This method has a natural interpretation in which we can think of it as approximating the function f via a linear function that is tangent to f at the current guess θ, solving for where that linear function equals to zero, and letting the next guess for θ be where that linear function is zero.

Here's a picture of the Newton's method in action:

In the leftmost figure, we see the function f plotted along with the line $y = 0$. We're trying to find θ so that $f(\theta) = 0$; the value of θ that achieves this is about 1.3. Suppose we initialized the algorithm with $\theta = 4.5$. Newton's method then fits a straight line tangent to f at $\theta = 4.5$, and solves for the where that line evaluates to 0. (Middle figure.) This give us the next guess for θ, which is about 2.8. The rightmost figure shows the result of running one more iteration, which the updates θ to about 1.8. After a few more iterations, we rapidly approach $\theta = 1.3$.

Newton's method gives a way of getting to $f(\theta) = 0$. What if we want to use it to maximize some function ℓ? The maxima of ℓ correspond to points where its first derivative $\ell'(\theta)$ is zero. So, by letting $f(\theta) = \ell'(\theta)$, we can use the same algorithm to maximize ℓ, and we obtain update rule:

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}.$$

(Something to think about: How would this change if we wanted to use Newton's method to minimize rather than maximize a function?)

Lastly, in our logistic regression setting, θ is vector-valued, so we need to generalize Newton's method to this setting. The generalization of Newton's method to this multidimensional setting (also called the Newton-Raphson method) is given by

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta).$$

Here, $\nabla_\theta \ell(\theta)$ is, as usual, the vector of partial derivatives of $\ell(\theta)$ with respect to the θ_i's; and H is an d-by-d matrix (actually, $d{+}1{-}by{-}$d+1, assuming that we include the intercept term) called the **Hessian**, whose entries are given by

$$H_{ij} = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}.$$

Newton's method typically enjoys faster convergence than (batch) gradient descent, and requires many fewer iterations to get very close to the minimum. One iteration of Newton's can, however, be more expensive than one iteration of gradient descent, since it requires finding and inverting an d-by-d Hessian; but so long as d is not too large, it is usually much faster overall. When Newton's method is applied to maximize the logistic regression log likelihood function $\ell(\theta)$, the resulting method is also called **Fisher scoring**.