8 *Logistic regression*

8.1 Introduction

One way to build a probabilistic classifier is to create a joint model of the form $p(y, \mathbf{x})$ and then to condition on $\mathbf{x}$, thereby deriving $p(y|\mathbf{x})$. This is called the generative approach. An alternative approach is to fit a model of the form $p(y|\mathbf{x})$ directly. This is called the **discriminative** approach, and is the approach we adopt in this chapter. In particular, we will assume discriminative models which are linear in the parameters. This will turn out to significantly simplify model fitting, as we will see. In Section 8.6, we compare the generative and discriminative approaches, and in later chapters, we will consider non-linear and non-parametric discriminative models.

8.2 Model specification

As we discussed in Section 1.4.6, logistic regression corresponds to the following binary classification model:

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^T\mathbf{x})) \tag{8.1}$$

A 1d example is shown in Figure 1.19(b). Logistic regression can easily be extended to higher-dimensional inputs. For example, Figure 8.1 shows plots of $p(y = 1|\mathbf{x}, \mathbf{w}) = \text{sigm}(\mathbf{w}^T\mathbf{x})$ for 2d input and different weight vectors $\mathbf{w}$. If we threshold these probabilities at 0.5, we induce a linear decision boundary, whose normal (perpendicular) is given by $\mathbf{w}$.

8.3 Model fitting

In this section, we discuss algorithms for estimating the parameters of a logistic regression model.

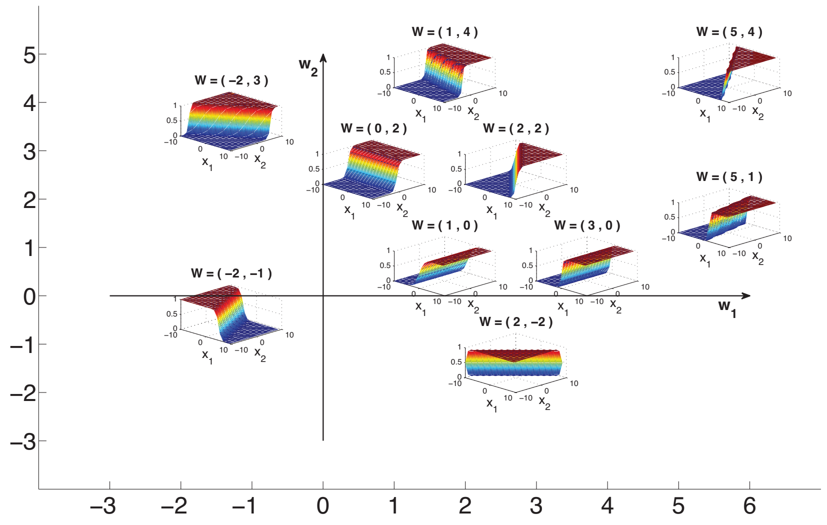

Figure 8.1 Plots of $\text{sigm}(w_1 x_1 + w_2 x_2)$. Here $\mathbf{w} = (w_1, w_2)$ defines the normal to the decision boundary. Points to the right of this have $\text{sigm}(\mathbf{w}^T\mathbf{x}) > 0.5$, and points to the left have $\text{sigm}(\mathbf{w}^T\mathbf{x}) < 0.5$. Based on Figure 39.3 of (MacKay 2003). Figure generated by `sigmoidplot2D`.

8.3.1 MLE

The negative log-likelihood for logistic regression is given by

$$\text{NLL}(\mathbf{w}) = -\sum_{i=1}^{N} \log[\mu_i^{\mathbb{I}(y_i=1)} \times (1-\mu_i)^{\mathbb{I}(y_i=0)}] \quad (8.2)$$

$$= -\sum_{i=1}^{N} [y_i \log \mu_i + (1-y_i)\log(1-\mu_i)] \quad (8.3)$$

This is also called the **cross-entropy** error function (see Section 2.8.2).

Another way of writing this is as follows. Suppose $\tilde{y}_i \in \{-1,+1\}$ instead of $y_i \in \{0,1\}$. We have $p(y=1) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$ and $p(y=1) = \frac{1}{1+\exp(+\mathbf{w}^T\mathbf{x})}$. Hence

$$NLL(\mathbf{w}) = \sum_{i=1}^{N} \log(1+\exp(-\tilde{y}_i \mathbf{w}^T\mathbf{x}_i)) \quad (8.4)$$

Unlike linear regression, we can no longer write down the MLE in closed form. Instead, we need to use an optimization algorithm to compute it. For this, we need to derive the gradient and Hessian.

In the case of logistic regression, one can show (Exercise 8.3) that the gradient and Hessian

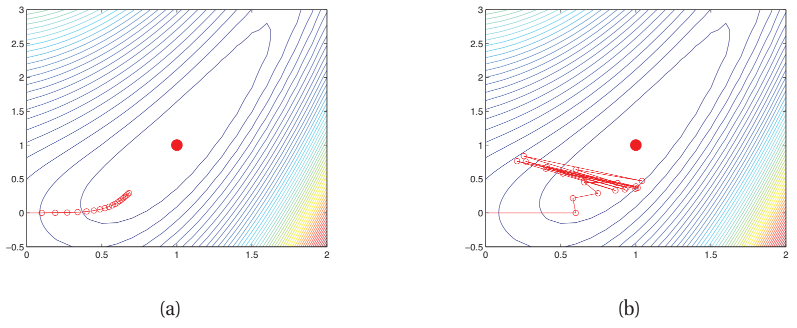

Figure 8.2 Gradient descent on a simple function, starting from $(0, 0)$, for 20 steps, using a fixed learning rate (step size) η. The global minimum is at $(1, 1)$. (a) $\eta = 0.1$. (b) $\eta = 0.6$. Figure generated by `steepestDescentDemo`.

of this are given by the following

$$\mathbf{g} = \frac{d}{d\mathbf{w}} f(\mathbf{w}) = \sum_i (\mu_i - y_i)\mathbf{x}_i = \mathbf{X}^T(\boldsymbol{\mu} - \mathbf{y}) \tag{8.5}$$

$$\mathbf{H} = \frac{d}{d\mathbf{w}} \mathbf{g}(\mathbf{w})^T = \sum_i (\nabla_{\mathbf{w}} \mu_i)\mathbf{x}_i^T = \sum_i \mu_i(1 - \mu_i)\mathbf{x}_i\mathbf{x}_i^T \tag{8.6}$$

$$= \mathbf{X}^T\mathbf{S}\mathbf{X} \tag{8.7}$$

where $\mathbf{S} \triangleq \text{diag}(\mu_i(1 - \mu_i))$. One can also show (Exercise 8.3) that $\mathbf{H}$ is positive definite. Hence the NLL is convex and has a unique global minimum. Below we discuss some methods for finding this minimum.

8.3.2 Steepest descent

Perhaps the simplest algorithm for unconstrained optimization is **gradient descent**, also known as **steepest descent**. This can be written as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k \tag{8.8}$$

where η_k is the **step size** or **learning rate**. The main issue in gradient descent is: how should we set the step size? This turns out to be quite tricky. If we use a constant learning rate, but make it too small, convergence will be very slow, but if we make it too large, the method can fail to converge at all. This is illustrated in Figure 8.2. where we plot the following (convex) function

$$f(\boldsymbol{\theta}) = 0.5(\theta_1^2 - \theta_2)^2 + 0.5(\theta_1 - 1)^2, \tag{8.9}$$

We arbitrarily decide to start from $(0, 0)$. In Figure 8.2(a), we use a fixed step size of $\eta = 0.1$; we see that it moves slowly along the valley. In Figure 8.2(b), we use a fixed step size of $\eta = 0.6$; we see that the algorithm starts oscillating up and down the sides of the valley and never converges to the optimum.

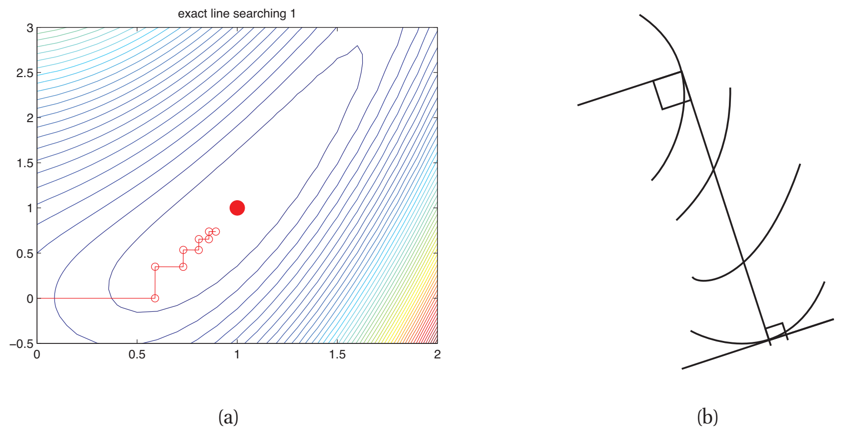

Figure 8.3 (a) Steepest descent on the same function as Figure 8.2, starting from $(0, 0)$, using line search. Figure generated by `steepestDescentDemo`. (b) Illustration of the fact that at the end of a line search (top of picture), the local gradient of the function will be perpendicular to the search direction. Based on Figure 10.6.1 of (Press et al. 1988).

Let us develop a more stable method for picking the step size, so that the method is guaranteed to converge to a local optimum no matter where we start. (This property is called **global convergence**, which should not be confused with convergence to the global optimum!) By Taylor's theorem, we have

$$f(\boldsymbol{\theta} + \eta \mathbf{d}) \approx f(\boldsymbol{\theta}) + \eta \mathbf{g}^T \mathbf{d} \tag{8.10}$$

where $\mathbf{d}$ is our descent direction. So if η is chosen small enough, then $f(\boldsymbol{\theta} + \eta \mathbf{d}) < f(\boldsymbol{\theta})$, since the gradient will be negative. But we don't want to choose the step size η too small, or we will move very slowly and may not reach the minimum. So let us pick η to minimize

$$\phi(\eta) = f(\boldsymbol{\theta}_k + \eta \mathbf{d}_k) \tag{8.11}$$

This is called **line minimization** or **line search**. There are various methods for solving this 1d optimization problem; see (Nocedal and Wright 2006) for details.

Figure 8.3(a) demonstrates that line search does indeed work for our simple problem. However, we see that the steepest descent path with exact line searches exhibits a characteristic **zig-zag** behavior. To see why, note that an exact line search satisfies $\eta_k = \arg\min_{\eta>0} \phi(\eta)$. A necessary condition for the optimum is $\phi'(\eta) = 0$. By the chain rule, $\phi'(\eta) = \mathbf{d}^T \mathbf{g}$, where $\mathbf{g} = f'(\boldsymbol{\theta} + \eta \mathbf{d})$ is the gradient at the end of the step. So we either have $\mathbf{g} = \mathbf{0}$, which means we have found a stationary point, or $\mathbf{g} \perp \mathbf{d}$, which means that exact search stops at a point where the local gradient is perpendicular to the search direction. Hence consecutive directions will be orthogonal (see Figure 8.3(b)). This explains the zig-zag behavior.

One simple heuristic to reduce the effect of zig-zagging is to add a **momentum** term, $(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$, as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k + \mu_k (\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}) \tag{8.12}$$

where $0 \leq \mu_k \leq 1$ controls the importance of the momentum term. In the optimization community, this is known as the **heavy ball method** (see e.g., (Bertsekas 1999)).

An alternative way to minimize “zig-zagging” is to use the method of **conjugate gradients** (see e.g., (Nocedal and Wright 2006, ch 5) or (Golub and van Loan 1996, Sec 10.2)). This is the method of choice for quadratic objectives of the form $f(\boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{A} \boldsymbol{\theta}$, which arise when solving linear systems. However, non-linear CG is less popular.

8.3.3 Newton’s method

Algorithm 8.1: Newton’s method for minimizing a strictly convex function

```
1 Initialize $\boldsymbol{\theta}_0$;
2 for $k = 1, 2, \ldots$ until convergence do
3     Evaluate $\mathbf{g}_k = \nabla f(\boldsymbol{\theta}_k)$;
4     Evaluate $\mathbf{H}_k = \nabla^2 f(\boldsymbol{\theta}_k)$;
5     Solve $\mathbf{H}_k \mathbf{d}_k = -\mathbf{g}_k$ for $\mathbf{d}_k$;
6     Use line search to find stepsize $\eta_k$ along $\mathbf{d}_k$;
7     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k$;
```

One can derive faster optimization methods by taking the curvature of the space (i.e., the Hessian) into account. These are called **second order** optimization metods. The primary example is **Newton’s algorithm**. This is an iterative algorithm which consists of updates of the form

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{H}_k^{-1} \mathbf{g}_k \tag{8.13}$$

The full pseudo-code is given in Algorithm 2.

This algorithm can be derived as follows. Consider making a second-order Taylor series approximation of $f(\boldsymbol{\theta})$ around $\boldsymbol{\theta}_k$:

$$f_{quad}(\boldsymbol{\theta}) = f_k + \mathbf{g}_k^T (\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \mathbf{H}_k (\boldsymbol{\theta} - \boldsymbol{\theta}_k) \tag{8.14}$$

Let us rewrite this as

$$f_{quad}(\boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{A} \boldsymbol{\theta} + \mathbf{b}^T \boldsymbol{\theta} + c \tag{8.15}$$

where

$$\mathbf{A} = \frac{1}{2} \mathbf{H}_k, \;\; \mathbf{b} = \mathbf{g}_k - \mathbf{H}_k \boldsymbol{\theta}_k, \;\; c = f_k - \mathbf{g}_k^T \boldsymbol{\theta}_k + \frac{1}{2} \boldsymbol{\theta}_k^T \mathbf{H}_k \boldsymbol{\theta}_k \tag{8.16}$$

The minimum of f_{quad} is at

$$\boldsymbol{\theta} = -\frac{1}{2} \mathbf{A}^{-1} \mathbf{b} = \boldsymbol{\theta}_k - \mathbf{H}_k^{-1} \mathbf{g}_k \tag{8.17}$$

Thus the Newton step $\mathbf{d}_k = -\mathbf{H}_k^{-1} \mathbf{g}_k$ is what should be added to $\boldsymbol{\theta}_k$ to minimize the second order approximation of f around $\boldsymbol{\theta}_k$. See Figure 8.4(a) for an illustration.

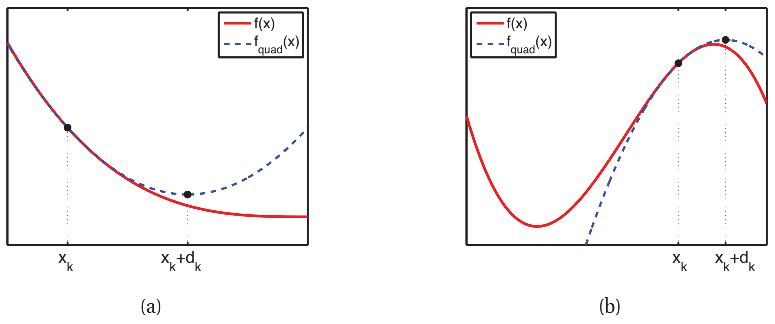

Figure 8.4 Illustration of Newton's method for minimizing a 1d function. (a) The solid curve is the function $f(x)$. The dotted line $f_{quad}(x)$ is its second order approximation at x_k. The Newton step d_k is what must be added to x_k to get to the minimum of $f_{quad}(x)$. Based on Figure 13.4 of (Vandenberghe 2006). Figure generated by `newtonsMethodMinQuad`. (b) Illustration of Newton's method applied to a nonconvex function. We fit a quadratic around the current point x_k and move to its stationary point, $x_{k+1} = x_k + d_k$. Unfortunately, this is a local maximum, not minimum. This means we need to be careful about the extent of our quadratic approximation. Based on Figure 13.11 of (Vandenberghe 2006). Figure generated by `newtonsMethodNonConvex`.

In its simplest form (as listed), Newton's method requires that $\mathbf{H}_k$ be positive definite, which will hold if the function is strictly convex. If not, the objective function is not convex, then $\mathbf{H}_k$ may not be positive definite, so $\mathbf{d}_k = -\mathbf{H}_k^{-1}\mathbf{g}_k$ may not be a descent direction (see Figure 8.4(b) for an example). In this case, one simple strategy is to revert to steepest descent, $\mathbf{d}_k = -\mathbf{g}_k$. The **Levenberg Marquardt** algorithm is an adaptive way to blend between Newton steps and steepest descent steps. This method is widely used when solving nonlinear least squares problems. An alternative approach is this: Rather than computing $\mathbf{d}_k = -\mathbf{H}_k^{-1}\mathbf{g}_k$ directly, we can solve the linear system of equations $\mathbf{H}_k\mathbf{d}_k = -\mathbf{g}_k$ for $\mathbf{d}_k$ using conjugate gradient (CG). If $\mathbf{H}_k$ is not positive definite, we can simply truncate the CG iterations as soon as negative curvature is detected; this is called **truncated Newton**.

8.3.4 Iteratively reweighted least squares (IRLS)

Let us now apply Newton's algorithm to find the MLE for binary logistic regression. The Newton update at iteration $k+1$ for this model is as follows (using $\eta_k = 1$, since the Hessian is exact):

$$\begin{aligned}
\mathbf{w}_{k+1} &= \mathbf{w}_k - \mathbf{H}^{-1}\mathbf{g}_k && (8.18)\\
&= \mathbf{w}_k + (\mathbf{X}^T\mathbf{S}_k\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y}-\boldsymbol{\mu}_k) && (8.19)\\
&= (\mathbf{X}^T\mathbf{S}_k\mathbf{X})^{-1}\left[(\mathbf{X}^T\mathbf{S}_k\mathbf{X})\mathbf{w}_k + \mathbf{X}^T(\mathbf{y}-\boldsymbol{\mu}_k)\right] && (8.20)\\
&= (\mathbf{X}^T\mathbf{S}_k\mathbf{X})^{-1}\mathbf{X}^T\left[\mathbf{S}_k\mathbf{X}\mathbf{w}_k + \mathbf{y}-\boldsymbol{\mu}_k\right] && (8.21)\\
&= (\mathbf{X}^T\mathbf{S}_k\mathbf{X})^{-1}\mathbf{X}^T\mathbf{S}_k\mathbf{z}_k && (8.22)
\end{aligned}$$

where we have defined the **working response** as

$$\mathbf{z}_k \triangleq \mathbf{X}\mathbf{w}_k + \mathbf{S}_k^{-1}(\mathbf{y}-\boldsymbol{\mu}_k) \qquad (8.23)$$

Equation 8.22 is an example of a **weighted least squares problem**, which is a minimizer of

$$\sum_{i=1}^{N} S_{ki}(z_{ki} - \mathbf{w}^T \mathbf{x}_i)^2 \tag{8.24}$$

Since $\mathbf{S}_k$ is a diagonal matrix, we can rewrite the targets in component form (for each case $i = 1 : N$) as

$$z_{ki} = \mathbf{w}_k^T \mathbf{x}_i + \frac{y_i - \mu_{ki}}{\mu_{ki}(1 - \mu_{ki})} \tag{8.25}$$

This algorithm is known as **iteratively reweighted least squares** or **IRLS** for short, since at each iteration, we solve a weighted least squares problem, where the weight matrix $\mathbf{S}_k$ changes at each iteration. See Algorithm 10 for some pseudocode.

Algorithm 8.2: Iteratively reweighted least squares (IRLS)

```
1  $\mathbf{w} = \mathbf{0}_D$;
2  $w_0 = \log(\overline{y}/(1 - \overline{y}))$;
3  repeat
4      $\eta_i = w_0 + \mathbf{w}^T \mathbf{x}_i$;
5      $\mu_i = \text{sigm}(\eta_i)$;
6      $s_i = \mu_i(1 - \mu_i)$ ;
7      $z_i = \eta_i + \frac{y_i - \mu_i}{s_i}$ ;
8      $\mathbf{S} = \text{diag}(s_{1:N})$ ;
9      $\mathbf{w} = (\mathbf{X}^T \mathbf{S} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{S} \mathbf{z}$;
10 until converged;
```

8.3.5 Quasi-Newton (variable metric) methods

The mother of all second-order optimization algorithm is Newton's algorithm, which we discussed in Section 8.3.3. Unfortunately, it may be too expensive to compute $\mathbf{H}$ explicitly. **Quasi-Newton** methods iteratively build up an approximation to the Hessian using information gleaned from the gradient vector at each step. The most common method is called **BFGS** (named after its inventors, Broyden, Fletcher, Goldfarb and Shanno), which updates the approximation to the Hessian $\mathbf{B}_k \approx \mathbf{H}_k$ as follows:

$$\begin{aligned} \mathbf{B}_{k+1} &= \mathbf{B}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{(\mathbf{B}_k \mathbf{s}_k)(\mathbf{B}_k \mathbf{s}_k)^T}{\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k} && (8.26) \\ \mathbf{s}_k &= \boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1} && (8.27) \\ \mathbf{y}_k &= \mathbf{g}_k - \mathbf{g}_{k-1} && (8.28) \end{aligned}$$

This is a rank-two update to the matrix, and ensures that the matrix remains positive definite (under certain restrictions on the step size). We typically start with a diagonal approximation, $\mathbf{B}_0 = \mathbf{I}$. Thus BFGS can be thought of as a "diagonal plus low-rank" approximation to the Hessian.

Alternatively, BFGS can iteratively update an approximation to the inverse Hessian, $\mathbf{C}_k \approx \mathbf{H}_k^{-1}$, as follows:

$$\mathbf{C}_{k+1} = \left(\mathbf{I} - \frac{\mathbf{s}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) \mathbf{C}_k \left(\mathbf{I} - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) + \frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} \tag{8.29}$$

Since storing the Hessian takes $O(D^2)$ space, for very large problems, one can use **limited memory BFGS**, or **L-BFGS**, where $\mathbf{H}_k$ or $\mathbf{H}_k^{-1}$ is approximated by a diagonal plus low rank matrix. In particular, the product $\mathbf{H}_k^{-1}\mathbf{g}_k$ can be obtained by performing a sequence of inner products with $\mathbf{s}_k$ and $\mathbf{y}_k$, using only the m most recent $(\mathbf{s}_k, \mathbf{y}_k)$ pairs, and ignoring older information. The storage requirements are therefore $O(mD)$. Typically $m \sim 20$ suffices for good performance. See (Nocedal and Wright 2006, p177) for more information. L-BFGS is often the method of choice for most unconstrained smooth optimization problems that arise in machine learning (although see Section 8.5).

8.3.6 ℓ_2 regularization

Just as we prefer ridge regression to linear regression, so we should prefer MAP estimation for logistic regression to computing the MLE. In fact, regularization is important in the classification setting even if we have lots of data. To see why, suppose the data is linearly separable. In this case, the MLE is obtained when $||\mathbf{w}|| \to \infty$, corresponding to an infinitely steep sigmoid function, $\mathbb{I}(\mathbf{w}^T\mathbf{x} > w_0)$, also known as a **linear threshold unit**. This assigns the maximal amount of probability mass to the training data. However, such a solution is very brittle and will not generalize well.

To prevent this, we can use ℓ_2 regularization, just as we did with ridge regression. We note that the new objective, gradient and Hessian have the following forms:

$$f'(\mathbf{w}) = \text{NLL}(\mathbf{w}) + \lambda \mathbf{w}^T\mathbf{w} \tag{8.30}$$

$$\mathbf{g}'(\mathbf{w}) = \mathbf{g}(\mathbf{w}) + \lambda \mathbf{w} \tag{8.31}$$

$$\mathbf{H}'(\mathbf{w}) = \mathbf{H}(\mathbf{w}) + \lambda \mathbf{I} \tag{8.32}$$

It is a simple matter to pass these modified equations into any gradient-based optimizer.

8.3.7 Multi-class logistic regression

Now we consider **multinomial logistic regression**, sometimes called a **maximum entropy classifier**. This is a model of the form

$$p(y = c|\mathbf{x}, \mathbf{W}) = \frac{\exp(\mathbf{w}_c^T\mathbf{x})}{\sum_{c'=1}^{C} \exp(\mathbf{w}_{c'}^T\mathbf{x})} \tag{8.33}$$

A slight variant, known as a **conditional logit model**, normalizes over a different set of classes for each data case; this can be useful for modeling choices that users make between different sets of items that are offered to them.

Let us now introduce some notation. Let $\mu_{ic} = p(y_i = c|\mathbf{x}_i, \mathbf{W}) = \mathcal{S}(\boldsymbol{\eta}_i)_c$, where $\boldsymbol{\eta}_i = \mathbf{W}^T\mathbf{x}_i$ is a $C \times 1$ vector. Also, let $y_{ic} = \mathbb{I}(y_i = c)$ be the one-of-C encoding of y_i; thus $\mathbf{y}_i$ is a bit vector, in which the c'th bit turns on iff $y_i = c$. Following (Krishnapuram et al. 2005), let us