Section 9. Predicting a discrete response with a continuous predictor

Unconstrained optimization. Throughout this section we assume there is a function $f(\theta)$ to be minimized without any constraints on $\theta \in \mathbb{R}^d$. Typical examples in statistics would be the negative log-likelihood or negative log-posterior, where θ is a parameter vector. Perhaps the simplest algorithm for unconstrained optimization is gradient descent, also known as steepest descent.

Definition 9.1. A *descent direction* at θ is a vector $\mathbf{d}$ such that

$$f(\theta + \eta \mathbf{d}) < f(\theta)$$

for all sufficiently small $\eta > 0$. A *descent algorithm* is an iterative procedure of the form

$$\theta_{k+1} = \theta_k + \eta_k \mathbf{d}_k$$

together with some prescription for choosing η_k and $\mathbf{d}_k$ based on θ_k and, of course, behavior of $f(\theta)$ in a neighborhood of θ_k.

Generally if $\mathbf{d}$ is some vector such that

$$\mathbf{g} \cdot \mathbf{d} < 0 \tag{9.1}$$

where $\mathbf{g} = \nabla f(\theta)$, then by Taylor's theorem

$$f(\theta + \eta \mathbf{d}) = f(\theta) + \eta \underbrace{\mathbf{g} \cdot \mathbf{d}}_{\text{negative}} + O(\eta^2)$$

so for this reason, (9.1) is an alternative characterization of a descent direction.

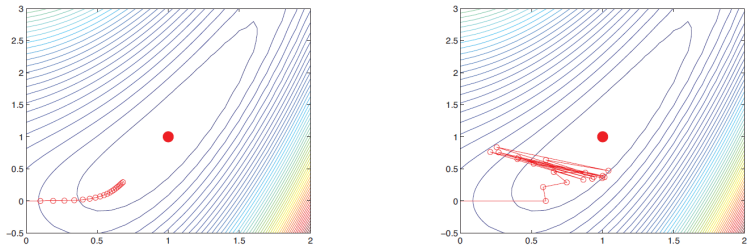
The most commonly used descent direction is the negative gradient,

$$-\mathbf{g}_k = -\nabla f(\theta_k)$$

With this choice, the associated descent algorithm is

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$

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. In particular, setting $\eta_k =$ constant is usually not a good idea: if the constant is small, convergence will be very slow, and if the constant is large, the method can fail to converge at all. This is illustrated in Fig. 1.



(a) (b)

FIGURE 1. Gradient descent on

$$f(\theta) = 0.5(\theta_1^2 - \theta_2)^2 + 0.5(\theta_1 - 1)^2$$

starting from (0, 0), for 20 steps, using a fixed step size η. The global minimum is at (1, 1). (a) $\eta = 0.1$. (b) $\eta = 0.6$.

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!)

Suppose we are at θ_k and we have chosen a descent direction $\mathbf{d}_k$. We can then view picking the best η_k as a one-dimensional optimization problem: minimize $\phi(\eta)$ where

$$\phi(\eta) = f(\theta_k + \eta \mathbf{d}_k)\,. \tag{9.2}$$

Definition 9.2. Any method which finds η for which $f(\theta_k + \eta \mathbf{d}_k) < f(\theta_k)$ is called a **line search method**. Popular line search methods include backtracking line search and More-Thuente line search; see Nocedal and Wright (2006) for details. **Exact line search** refers to finding (as close as possible) the exact minimum of (9.2).

Usually something "exact" would be preferred to something approximate, but that isn't really true for line search methods! Exact line search is usually not efficient in the sense that fully optimizing the function along a line that doesn't even contain the true optimum is less efficient than using your cpu cycles to compute the next search direction. Moreover, even with a very fast cpu, you wouldn't necessarily want to exactly minimize $\phi(\eta)$: the steepest descent path with exact line-search exhibits a characteristic zig-zag behavior. To see why, note that a necessary condition for the optimum is $\phi'(\eta) = 0$. By

the chain rule, $\phi'(\eta) = \mathbf{d}^T\mathbf{g}$, where $\mathbf{g} = f'(\theta + \eta\mathbf{d})$ is the gradient at the end of the step. So we either have $\mathbf{g} = 0$, 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. This explains the zig-zag behavior, and is another reason exact line search is not very efficient.

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 methods. The primary example is Newton's algorithm. This is an iterative algorithm which consists of updates of the form

$$\theta_{k+1} = \theta_k - \eta_k \mathbb{H}_k^{-1}\mathbf{g}_k \,.$$

where η_k is determined by line-search. Typically if you possess the exact Hessian inverse $\mathbb{H}_k^{-1}$ then using $\eta_k = 1$ is fine, but if you're using some approximation to the Hessian (which most actual methods do), then choosing η_k by line search can still be helpful.

Newton's algorithm can be derived as follows. Consider making a second-order Taylor series approximation of $f(\theta)$ around θ_k:

$$f_{\text{quad}}(\theta) = f_k + \mathbf{g}_k^T(\theta - \theta_k) + \frac{1}{2}(\theta - \theta_k)^T\mathbb{H}_k(\theta - \theta_k)$$

Let us rewrite this as

$$f_{\text{quad}}(\theta) = \frac{1}{2}\theta^T A\theta + b^T\theta + c$$

where

$$A = \mathbb{H}_k, b = \mathbf{g}_k - \mathbb{H}_k\theta_k, c = f_k - \mathbf{g}_k^T\theta_k + \frac{1}{2}\theta_k^T\mathbb{H}_k\theta_k$$

The minimum of f_{quad} is at

$$\theta = -A^{-1}b = \theta_k - \mathbb{H}_k^{-1}\mathbf{g}_k$$

Thus the Newton step $\mathbf{d}_k = -\mathbb{H}_k^{-1}\mathbf{g}_k$ is what should be added to θ_k to minimize the second order approximation of f around θ_k. As long as $\mathbb{H}_k$ has no negative eigenvalues,

$$\langle \mathbf{d}_k, \mathbf{g}_k \rangle = -\langle \mathbf{g}_k, \mathbb{H}_k\mathbf{g}_k \rangle < 0$$

and hence the Newton step $\mathbf{d}_k = -\mathbb{H}_k^{-1}\mathbf{g}_k$ is a descent direction. Positive definiteness of $\mathbb{H}_k$ will of course hold if the function is strictly convex. Otherwise, $\mathbf{d}_k = -\mathbb{H}_k\mathbf{g}_k$ may not be a descent direction. 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: rather than computing $\mathbf{d}_k = -\mathbb{H}_k^{-1}\mathbf{g}_k$ directly, we can solve the linear system of equations $\mathbb{H}_k\mathbf{d}_k = -\mathbf{g}_k$ for $\mathbf{d}_k$ using conjugate gradient (CG). If $\mathbb{H}_k$ is not positive definite, we truncate the CG iterations as soon as negative curvature is detected; this is called truncated Newton.

It may be too expensive to compute $\mathbb{H}$ 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 $B_k \approx H_k$ as follows:

$$\begin{aligned} B_{k+1} &= B_k + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k} - \frac{(B_k\mathbf{s}_k)(B_k\mathbf{s}_k)^T}{\mathbf{s}_k^T B_k\mathbf{s}_k} \\ \mathbf{s}_k &= \theta_k - \theta_{k-1} \\ \mathbf{y}_k &= \mathbf{g}_k - \mathbf{g}_{k-1} \end{aligned}$$

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, $B_0 = I$. 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, $C_k \approx \mathbb{H}_k^{-1}$ as follows:

$$C_{k+1} = \left(I - \frac{\mathbf{s}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right) C_k \left(I - \frac{\mathbf{y}_k\mathbf{s}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}\right) + \frac{\mathbf{s}_k\mathbf{s}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}$$

Since storing the Hessian takes $O(D^2)$ space, for very large problems, one can use limited memory BFGS, or L-BFGS, where $\mathbb{H}_k$ or $\mathbb{H}_k^{-1}$ is approximated by a diagonal plus low rank matrix and the product $\mathbb{H}_k^{-1}\mathbf{g}_k$ 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, p.177) for more information.

L-BFGS is often the method of choice for most unconstrained smooth optimization problems that arise in machine learning, and like pretty much

every optimization algorithm known to man, free open-source implementations with GPL type licenses exist on the internet.

Logistic Regression. We begin with some review. Suppose we toss a coin n times. Let $N_H \in \{0, \ldots, n\}$ be the number of heads. If the probability of heads is θ, then we say N_H has a binomial distribution, written as $N_H \sim \text{Bin}(n, \theta)$ where

$$\text{Bin}(k \,|\, n, \theta) = \binom{n}{k} \theta^k (1-\theta)^{n-k}.$$

The special case of $n = 1$ is called a Bernoulli distribution and will be written $\text{Ber}(k \,|\, \theta)$.

Now we want to know how to predict binary variables $y \in \{0, 1\}$, using data which may be continuous. The basic idea is that binary variables are naturally represented as having a Bernoulli distribution conditional on some probability $\theta \in [0, 1]$, and since θ is a continuous variable, we might have some hope of predicting it using other continuous variables, as long as we can deal with the fact that it's limited to the range [0,1].

This is a two-level hierarchical model, and it's easy to see how having some experience thinking about hierarchical models would have made it trivial to come up with this.

The model is thus

$$p(y \,|\, \mathbf{x}, \mathbf{w}) = \text{Ber}(y \,|\, \mu_{\mathbf{w}}(\mathbf{x}))$$
$$\mu_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}) \,.$$

where $\sigma(\eta)$ refers to the sigmoid function, also known as the logistic or logit function. This is defined as

$$\sigma(\eta) = \frac{1}{1 + \exp(-\eta)} = \frac{e^\eta}{e^\eta + 1} \tag{9.3}$$

The term "sigmoid" means S-shaped. Note that σ maps the whole real line to $[0, 1]$, which is necessary for the output to be interpreted as a probability. The scaled version is denoted

$$\sigma_\beta(\eta) = \sigma(\beta\eta)$$

This is sometimes useful because as $\beta \to \infty$ it approaches a step function.

Putting these two steps together we get

$$p(y \,|\, \mathbf{x}, \mathbf{w}) = \text{Ber}(y \,|\, \sigma(\mathbf{w}^T \mathbf{x})) \tag{9.4}$$

This is called logistic regression due to its similarity to linear regression. More generally one may include a "bias parameter" b so that

$$p(y = 1 \,|\, \mathbf{x}, \mathbf{w}) = \sigma(b + \mathbf{w}^T \mathbf{x}).$$

If we use σ_β and take the limit as $\beta \to \infty$ leads to a machine learning model which is equivalent to the historically important "perceptron."

The **decision boundary** is defined as the set $\{\mathbf{x} : p(y = 1 \,|\, \mathbf{x}, \mathbf{w}) = 0.5\}$. This is given by the hyperplane

$$b + \mathbf{x}^T \mathbf{w} = 0$$

On the side of the hyperplane for which $b + \mathbf{x}^T \mathbf{w} > 0$, inputs $\mathbf{x}$ are classified as 1s, and on the other side they are classified as 0s. The 'bias' parameter b simply shifts the decision boundary by a constant amount. The orientation of the decision boundary is determined by $\mathbf{w}$, the normal to the hyperplane.

To derive a model which is useful for making predictions, we have to first "train" the model, which just means performing inference on the parameters using whatever data we already have, hence don't need to make predictions about. Suppose that we have a data set consisting of N predictor-response pairs

$$D = \{(\mathbf{x}_i, y_i) \;:\; i = 1, \ldots, N\}.$$

Define the following, which is a function of $\mathbf{w}$ although we do not always denote its dependence explicitly:

$$\mu_i := \mu(\mathbf{x}_i) = \sigma(\mathbf{w}^T \mathbf{x}_i);$$

The negative log-likelihood for this data given the model (9.4) is

$$\begin{aligned} f(\mathbf{w}) &= -\sum_{i=1}^{N} \log \left[\mu_i^{\mathbb{I}(y_i=1)} \times (1 - \mu_i)^{\mathbb{I}(y_i=0)} \right] \\ &= -\sum_{i=1}^{N} \left[y_i \log \mu_i + (1 - y_i) \log(1 - \mu_i) \right] \end{aligned}$$

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.

$$\mathbf{g} = \nabla f(\mathbf{w}) = \sum_i (\mu_i - y_i)\mathbf{x}_i = X^T(\boldsymbol{\mu} - \mathbf{y}) \tag{9.5}$$

$$\begin{aligned} \mathbb{H} &= \nabla \mathbf{g}(\mathbf{w}) = \sum_i (\nabla_{\mathbf{w}} \mu_i)\mathbf{x}_i^T = \sum_i \mu_i(1-\mu_i)\mathbf{x}_i\mathbf{x}_i^T \\ &= X^T S X \end{aligned} \tag{9.6}$$

$$\text{where } S = \text{diag}(\mu_i(1-\mu_i)) \tag{9.7}$$

If X is full rank and $0 < \mu_i < 1$, then $\mathbb{H}$ is positive definite, and hence the negative log-likelihood is convex, by the second-order condition for convexity that we learned in an earlier lecture.

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):

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

where

$$S_k := \text{diag}\left(\mu_{ki}(1-\mu_{ki})\right), \quad \mu_{ki} = \sigma(\mathbf{w}_k^T \mathbf{x}_i)\,.$$

and we define the **working response** as

$$\mathbf{z}_k = X\mathbf{w}_k + S_k^{-1}(\mathbf{y} - \boldsymbol{\mu}_k)\,.$$

Note of course that S_k is diagonal, so computation of the working response doesn't actually necessitate matrix inversion. Note also that $(X^T S_k X)^{-1} X^T S_k \mathbf{z}_k$ is the familiar weighted least squares estimator for regressing response $\mathbf{z}_k$ on predictors X with weights S_k. In other words,

$$\mathbf{w}_{k+1} = \text{argmin}_{\mathbf{w}} \sum_{i=1}^N S_{ki}(z_{ki} - \mathbf{w}^T \mathbf{x}_i)^2$$

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 S_k changes at each iteration.

Even though reliable numerical implementations of Newton-type methods exist, in this case we benefited from working out the explicit form of the Newton update; this has enabled us to see the relationship with weighted least squares. The presumed existence of good software for doing something should never preclude us from trying to understand how that something is actually done under the covers.

It is natural to want to compute the full posterior over the parameters, $p(\mathbf{w} \mid D)$, for logistic regression models. This can be useful for any situation where we want to associate confidence intervals with our predictions. Unfortunately, unlike the linear regression case, this cannot be done exactly, since there is no convenient conjugate prior for logistic regression. We discuss one simple approximation below; other approaches include MCMC, variational inference, expectation propagation (Kuss and Rasmussen, 2005) etc. For simplicity, we stick to binary logistic regression in this lecture.

Given the posterior, we can compute credible intervals, perform hypothesis tests, etc., just as we did in the case of linear regression. But in machine learning, interest usually focuses on prediction. The posterior predictive distribution has the form

$$p(y \mid \mathbf{x}, D) = \int p(y \mid \mathbf{x}, \mathbf{w}) p(\mathbf{w} \mid D) d\mathbf{w}$$

Unfortunately this integral is intractable. The simplest approximation is called the "plug-in approximation," which means to use $p(y = 1 \mid \mathbf{x}, \hat{\mathbf{w}})$ where $\hat{\mathbf{w}}$ is a posterior-based point estimate of $\mathbf{w}$, such as the posterior mean. In this context, $\hat{\mathbf{w}}$ is called the "Bayes point," which is probably short for Bayesian point estimator. Of course, such a plug-in estimate underestimates the uncertainty. A better approach is to use a Monte Carlo approximation, as follows:

$$p(y = 1 \mid \mathbf{x}, D) \approx \frac{1}{S} \sum_{s=1}^{S} \sigma(\mathbf{w}_s^T \mathbf{x})$$

where $\mathbf{w}_s \sim p(\mathbf{w} \mid D)$ are samples from the posterior. (This technique can be trivially extended to the multi-class case.) If we have approximated the posterior using Monte Carlo, we can reuse these samples for prediction. If we made a Gaussian approximation to the posterior, we can draw independent samples from the Gaussian.

We could extend the above to predicting the outcomes of an experiment with K possible outcomes, where $K > 2$. For more than two classes, one

may use the softmax function

$$p(y = i \mid \mathbf{x}, \mathbf{w}) = \frac{e^{\mathbf{w}_i^T \mathbf{x} + b_i}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^T \mathbf{x} + b_j}}$$

References

Kuss, Malte and Carl Edward Rasmussen (2005). "Assessing approximate inference for binary Gaussian process classification". In: *The Journal of Machine Learning Research* 6, pp. 1679–1704.

Nocedal, Jorge and Stephen Wright (2006). *Numerical optimization*. Springer Science & Business Media.