13. Statistical Methods for Portfolio Management

13.1. **Descent Methods.**

13.1.1. *Descent Directions.* 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 13.1. A *descent direction* at θ is a vector $\boldsymbol{d}$ such that

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

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

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

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

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

$$\boldsymbol{g} \cdot \boldsymbol{d} < 0 \tag{13.1}$$

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

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

so for this reason, (13.1) is an alternative characterization of a descent direction.

The most obvious first attempt at finding a descent direction is to consider the negative gradient,

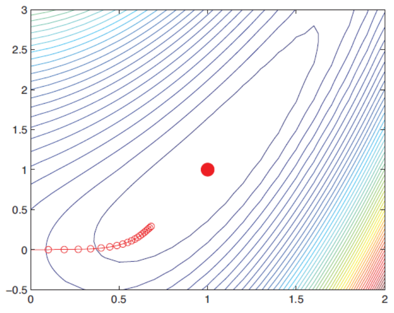
$$-\boldsymbol{g}_k = -\nabla f(\theta_k)$$

With this choice, the associated descent algorithm is

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

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. 13.1.


(a)

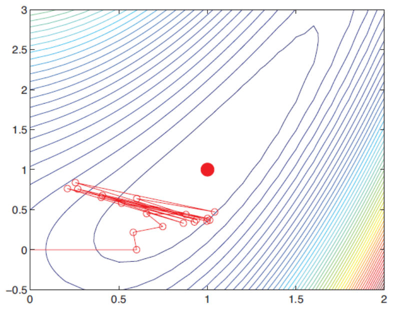

(b)

FIGURE 13.1. Gradient descent for $f(\theta) = 0.5(\theta_1^2 - \theta_2)^2 + 0.5(\theta_1 - 1)^2$ 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!)

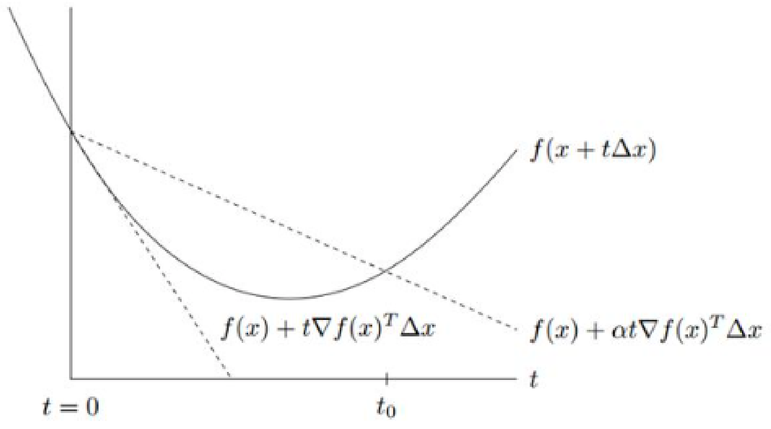
13.1.2. *Line Search.* Suppose we are at θ_k and we have chosen a descent direction $\boldsymbol{d}_k$. We can then view picking the best η_k as a one-dimensional optimization problem: minimize $\phi(\eta)$ where

$$\phi(\eta) = f(\theta_k + \eta \boldsymbol{d}_k)\,. \tag{13.2}$$

Definition 13.2. Any method which finds η for which $f(\theta_k + \eta \boldsymbol{d}_k) < f(\theta_k)$ is called a *line search method.* Popular line search methods include backtracking line search and Moré-Thuente line search (Moré and Thuente, 1994). *Exact line search* refers to finding (as closely as possible) the exact minimum of (13.2).

Backtracking line search is an exceedingly simple algorithm, trivial to implement. With parameters $\alpha \in (0, 1/2), \beta \in (0, 1)$, the algorithm is as follows: starting at $t = 1$, repeat $t := \beta t$ until

$$f(x + t\Delta x) < f(x) + \alpha t \nabla f(x)^T \Delta x$$



$f(x + t\Delta x)$
$f(x) + t\nabla f(x)^T \Delta x$
$f(x) + \alpha t \nabla f(x)^T \Delta x$
t
$t = 0$
t_0

Figure 9.1 *Backtracking line search.* The curve shows f, restricted to the line over which we search. The lower dashed line shows the linear extrapolation of f, and the upper dashed line has a slope a factor of α smaller. The backtracking condition is that f lies below the upper dashed line, *i.e.*, $0 \leq t \leq t_0$.

Example 13.1. Consider

$$f(x) = (1/2)(x_1^2 + \gamma x_2^2) \qquad (\gamma > 0)$$

with exact line search, starting at $x_0 = (\gamma, 1)$. One can exactly work out the iterates:

$$x_1^{(k)} = \gamma \left(\frac{\gamma - 1}{\gamma + 1} \right)^k \qquad x_2^{(k)} = \left(- \frac{\gamma - 1}{\gamma + 1} \right)^k$$

Hence we can see this is very slow if $\gamma \ll 1$ or $\gamma \gg 1$. Here is $\gamma = 10$:

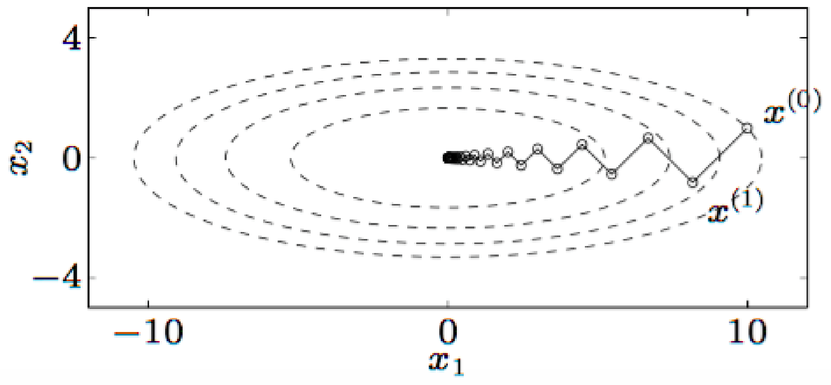


FIGURE 13.2. $\gamma = 10$.

A good expository reference for the various line search methods is Nocedal and Wright (2006).

Usually something "exact" would be preferred to something approximate, but that isn't necessarily 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, $\phi'(\eta) = \boldsymbol{d}^T\boldsymbol{g}$, where $\boldsymbol{g} = f'(\theta + \eta\boldsymbol{d})$ is the gradient at the end of the step. So we either have $\boldsymbol{g} = 0$, which means we have found a stationary point, or $\boldsymbol{g} \perp \boldsymbol{d}$, 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.

There are cases where exact line search performs very well (and outperforms other methods). Intuitively, this is the case when the "long narrow valley" problem doesn't present itself.

Example 13.2. Consider

$$f(x_1, x_2) = e^{x_1+3x_2-0.1} + e^{x_1-3x_2-0.1} + e^{-x_1-0.1}$$

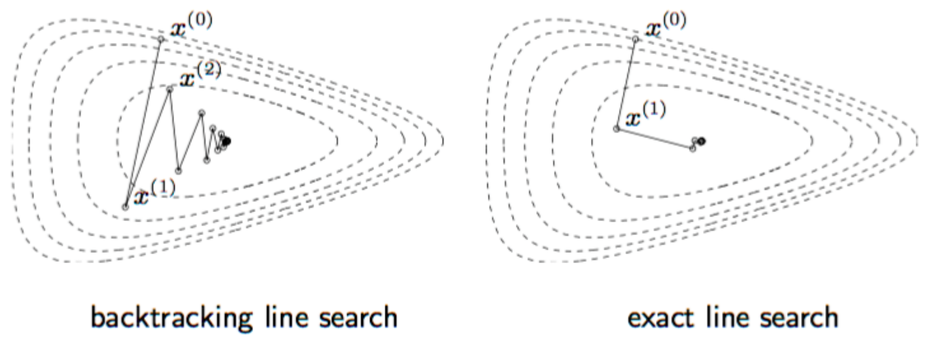


FIGURE 13.3. Backtracking vs. exact line search in a non-quadratic example.

13.1.3. *Second order optimization methods.* One can derive faster optimization methods by taking the curvature (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

$$\theta_{k+1} = \theta_k - \eta_k \boldsymbol{H}_k^{-1} \boldsymbol{g}_k \,. \tag{13.3}$$

where η_k is determined by line-search. We will show how to derive (13.3) momentarily but for now, take it as given. Typically if you possess the exact Hessian inverse $\boldsymbol{H}_k^{-1}$ 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:

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

Let us rewrite this as

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

where

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

The minimum of f_{quad} is at

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

Thus the Newton step $\boldsymbol{d}_k = -\boldsymbol{H}_k^{-1}\boldsymbol{g}_k$ is what should be added to θ_k to minimize the second order approximation of f around θ_k.

As long as $\boldsymbol{H}_k$ has no negative eigenvalues, then we know that

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

and hence the Newton step $\boldsymbol{d}_k = -\boldsymbol{H}_k^{-1}\boldsymbol{g}_k$ is a descent direction. Positive definiteness of $\boldsymbol{H}_k$ will of course hold if the function is strictly convex. Otherwise, $\boldsymbol{d}_k = -\boldsymbol{H}_k \boldsymbol{g}_k$ may not be a descent direction. In this case, one simple strategy is to revert to steepest descent, $\boldsymbol{d}_k = -\boldsymbol{g}_k$. The Levenberg-Marquardt algorithm is an adaptive way to blend between Newton steps and steepest descent steps, and is widely used when solving nonlinear least squares problems.

13.1.4. *Quasi-Newton Methods.* It may be too expensive to compute $\boldsymbol{H}$ 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:

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

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

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

Since storing the Hessian takes $O(D^2)$ space, for very large problems, one can use limited memory BFGS, or L-BFGS, where $\boldsymbol{H}_k$ or $\boldsymbol{H}_k^{-1}$ is approximated by a diagonal plus low rank matrix and the product $\boldsymbol{H}_k^{-1}\boldsymbol{g}_k$ can be obtained by performing a sequence of inner products with $\boldsymbol{s}_k$ and $\mathbf{y}_k$, using only the m most recent $(\boldsymbol{s}_k, \mathbf{y}_k)$ 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.

There is also an important generalization of L-BFGS to handle *bound constraints*, ie. constraints of the form

$$\ell_i \le x_i \le u_i,$$

where $\ell_i < u_i$ are real-valued bounds. These are sometimes referred to as *box constraints*, and arise very often in statistical parameter estimation problems. For example, one might want to enforce positivity for a variance parameter. The generalization, due to Byrd et al. (1995), is called L-BFGS-B and is available in R wth syntax

```
optim(x, f, method = 'L-BFGS-B', lower = ..., upper = ...)
```

and in SciPy where it is called via

```
scipy.optimize.minimize(fun, x0, args=(), method='L-BFGS-B').
```

13.2. **Example: 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

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

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

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

$$
\begin{aligned}
p(y \,|\, \mathbf{x}, \mathbf{w}) &= \mathrm{Ber}(y \,|\, \mu_{\mathbf{w}}(\mathbf{x})) \\
\mu_{\mathbf{w}}(\mathbf{x}) &= \sigma(\mathbf{w}^T \mathbf{x}) \,.
\end{aligned}
$$

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

$$
\sigma(\eta) = \frac{1}{1 + \exp(-\eta)} = \frac{e^\eta}{e^\eta + 1} \tag{13.4}
$$

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

$$
p(y \,|\, \mathbf{x}, \mathbf{w}) = \mathrm{Ber}(y \,|\, \sigma(\mathbf{w}^T \mathbf{x})) \tag{13.5}
$$

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 called the *perceptron*, perhaps the simplest (and historically first) neural network.

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

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

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 (13.5) is

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

where $\mathbb{I}$ denotes the indicator function.

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.

$$\begin{aligned} \boldsymbol{g} &= \nabla f(\mathbf{w}) = \sum_i (\mu_i - y_i)\mathbf{x}_i = X^T(\boldsymbol{\mu} - \mathbf{y}) && (13.6) \\ \boldsymbol{H} &= \nabla \boldsymbol{g}(\mathbf{w}) = \sum_i (\nabla_{\mathbf{w}} \mu_i)\mathbf{x}_i^T = \sum_i \mu_i(1-\mu_i)\mathbf{x}_i\mathbf{x}_i^T && (13.7) \\ &= X^T S X \\ & \quad \text{where } S = \operatorname{diag}(\mu_i(1-\mu_i)) && (13.8) \end{aligned}$$

If X is full rank and $0 < \mu_i < 1$, then $\boldsymbol{H}$ 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):

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

where

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

and we define the *working response* as

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

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

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

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 the procedure actually works.

13.3. **Model Selection.** Occam's (or Ockham's) razor is a principle attributed to the 14th century logician and Franciscan friar William of Ockham. Ockham was the village in Surrey where he was born. The general belief is that Ockham was fond of a saying along the lines *Pluralitas non est ponenda sine neccesitate*, which means "entities should not be multiplied unnecessarily". Many scientists have adopted or reinvented Occam's Razor, such as Newton:

> *We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances.*
>
> — Isaac Newton

The most useful statement of the principle for scientists is perhaps "given two competing theories that make exactly the same predictions, the simpler one is to be preferred".

It is a very common situation in data science problems that one does not know the model. One must guess some sort of model to proceed, but rarely is there a unique choice. Typically the same procedure, intuition, or thought process which led to the creation of the first model can just as well lead to others. A special case is *variable selection*, where a model of the form $y = f(\mathbf{x}; \theta)$ has been proposed, where $\mathbf{x} = (x_1, \ldots, x_k)$ and θ is a parameter vector, but it is typically not known whether all of the variables are helpful in forming out-of-sample predictions.

One approach is to use cross-validation to estimate the generalization error of all the candidate models. This approach is probably the most commonly used in practice, and so we will discuss it first, before moving on to more fully Bayesian methods.

Whenever there exists more than one candidate model, one can attempt to order the models by increasing *complexity*, where for now we leave "complexity" as a loosely defined concept and appeal to your intuition for what it means. Often, there is an explicit parameter which controls complexity. For example, you have seen in your homework and lectures that Lasso regression,

$$\hat{\beta}_\lambda = \operatorname*{argmin}_{\beta} \left[\|y - X\beta\|^2 + \lambda \, \|\beta\|_1 \right]$$

has the property that larger values of λ lead to more zeros in $\hat{\beta}_\lambda$, effectively de-selecting some of the variables, leading to a simpler model. So λ is inversely related to complexity: higher λ means *less* complexity.

In many cases, the complexity parameter may be discrete. In a previous lecture, we worked out in great detail (and even went through the source code) for a simple classifier based on the Gaussian mixture model. Our example included two mixture components, but the same methods could be trivially generalized to deal with an arbitrary number of mixture components, say n components. In this case, n is the model complexity parameter.

Overly complex models lead to suboptimal out-of-sample performance (as measured by forecast accuracy), as do overly simple models. The following figure shows an example of both kinds of badness:

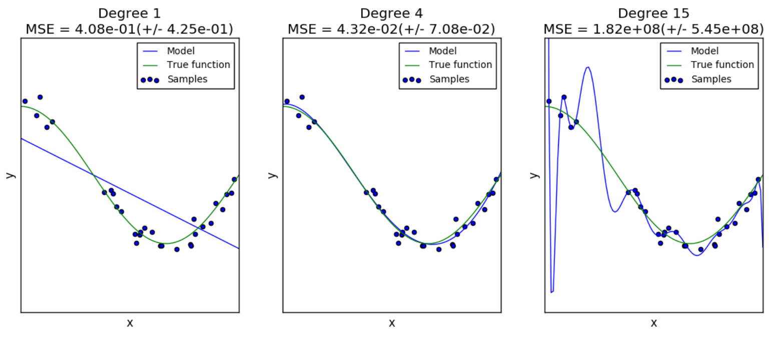


FIGURE 13.4. Overfitting and underfitting.

The overfit model is a better fit to the data, but will generalize poorly. Look at the wild fluctuations near the left side, for example. That's what the overfit model would predict if asked to predict for values between the points.

Consider a model of the form $y = f(x; \theta^{(\lambda)}; \lambda) + \epsilon$ where as discussed above, λ is a model complexity parameter. For a given level of complexity λ, there is a parameter space $\Theta^{(\lambda)}$ which contains candidate parameter vectors $\theta^{(\lambda)}$. Also, ϵ represents uncontrollable variance that is not explained by any model in the family we are considering. The parameter space can and usually does depend on the complexity; for example, if λ equals the number of mixture components, then the more components, the more parameters we will have to fit. Here $x \in \mathcal{X}$ is an element of the data space. Given parameters and given x, the model generates a prediction, $\hat{y} = f(x; \theta^{(\lambda)}; \lambda)$.

Given a family of models as above, here is how cross-validation works. Identify some subset of the data that you want to use for the purpose of figuring out how complex of a model you need. Call this the *training set* $\mathcal{T} \subset \mathcal{X}$. Let $\hat{\theta}^{(\lambda)}_A$ denote a fitted value of $\theta^{(\lambda)}$ using only data from subset A. Typically, this fitting is done via least squares, i.e.

$$\hat{\theta}^{(\lambda)}_A = \underset{\theta}{\operatorname{argmin}} \sum_{i:x_i \in A} \Big[y_i - f(x_i; \theta, \lambda) \Big]^2$$

Now suppose we are given a partition of the training set into k disjoint subsets, called *folds*,

$$\mathcal{T} = K_1 \cup K_2 \cup \ldots \cup K_k, \quad K_\alpha \cap K_\beta = \emptyset. \tag{13.9}$$

Also for convenience, define

$$K_{\neq\alpha} = \bigcup_{\substack{\gamma=1 \\ \gamma\neq\alpha}}^{k} K_\gamma = K_1 \cup \ldots \cup K_{\alpha-1} \cup K_{\alpha+1} \cup \ldots \cup K_k$$

to be the union of all except the α-th fold.

Define the *generalization error* on the α-th fold as

$$\mathrm{CV}_\alpha(\lambda) = \sum_{x_i \in K_\alpha} \left[y_i - f\left(x_i, \hat{\theta}^{(\lambda)}_{K_{\neq\alpha}}; \lambda\right)\right]^2 \tag{13.10}$$

Note that in the definition of $\mathrm{CV}_\alpha(\lambda)$, the parameter vector is fit on $K_{\neq\alpha}$, in other words, we do not use K_α when fitting $\hat{\theta}^{(\lambda)}_{K_{\neq\alpha}}$.

The *total generalization error* is just the sum of (13.10) over all folds:

$$\mathrm{CV}(\lambda) = \sum_{\alpha=1}^{k} \mathrm{CV}_\alpha(\lambda)$$

An example is as follows:

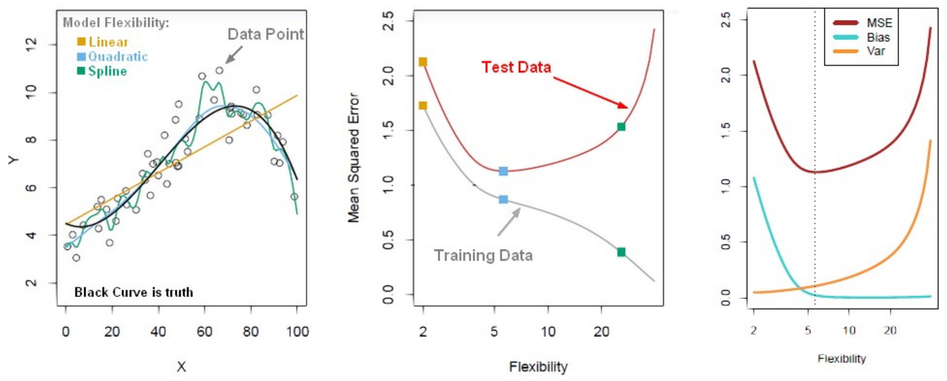


FIGURE 13.5. The bias-variance tradeoff.

Also note that in practice one would usually repeat the CV with various randomly selected partitions (13.9).

13.4. **Example: Lasso.** An equivalent way to describe Lasso is as a constrained problem:

$$\hat{\beta}_{\text{lasso}} = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 \right\} \tag{13.11}$$

$$\text{subject to: } \sum_{j=1}^{p} |\beta_j| \le t \tag{13.12}$$

Just as in ridge regression, we can re-parametrize the constant β_0 by standardizing the predictors; the solution for $\hat{\beta}_0$ is $\overline{y}$, and thereafter we fit a model without an intercept.

If t is chosen larger than $t_0 = \sum_1^p |\hat{\beta}_j|$ (where $\hat{\beta}_j$ denote the least squares estimates), then the lasso estimates are the $\hat{\beta}_j$. Hence a natural standardized parameter that is often used as the x-axis for plots is:

$$s = \frac{t}{\sum_{j=1}^{p} \left|\hat{\beta}_j\right|}$$

where again the denominator refers to least squares estimates. This standardized parameter then runs from 0 to 1.

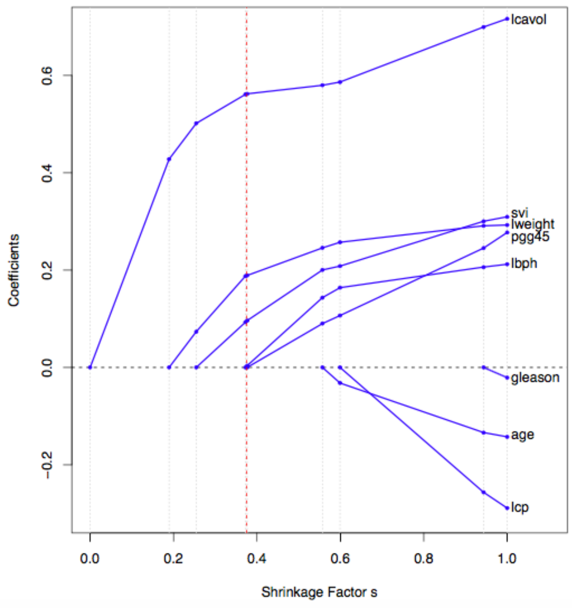


FIGURE 13.6. Profiles of lasso coefficients, as the tuning parameter t is varied. From Friedman, Hastie, and Tibshirani (2001)

The fitted coefficients in lasso depend strongly on λ. The appropriate value of λ to use can be determined by cross-validation.

In R, performing cross-validated elastic net regression is not much more complicated than OLS, and one can use the same data structure returned by the lm function as an input to elastic net. We give an example just below.

```
model <- lm(formla, data = mpan)
if(ENET) {
    X <- model.matrix(model)
    Y <- fitted(model) + residuals(model)
    CV <- cv.glmnet(X,Y)
    co.enet <- coef(CV, CV$lambda.min)
    rname <- row.names(co.enet)
    actual.coefs <- co.enet[rname[grep(pattern, rname)], ]
} else {
    co <- coef(model)
    actual.coefs <- co[names(co)[grep(pattern, names(co))]]
}
```

References

Byrd, Richard H et al. (1995). "A limited memory algorithm for bound constrained optimization". In: *SIAM Journal on Scientific Computing* 16.5, pp. 1190–1208.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani (2001). *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin.

Moré, Jorge J and David J Thuente (1994). "Line search algorithms with guaranteed sufficient decrease". In: *ACM Transactions on Mathematical Software (TOMS)* 20.3, pp. 286–307.

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