Solving One-dimensional Nonlinear Equations

Solve \( f(x) = c \). Assume \( f \) is continuous.

\((a,b)\) contains a solution if \( f(a) - c \) and \( f(b) - c \) have opposite signs.

Bisection search:
Find initial \((a,b)\) with \( f(a) > c \), \( f(b) < c \) (or the opposite). If \( f((a+b)/2) < c \), then new interval is \((a,(a+b)/2)\), otherwise \((a+b)/2, b)\) is.

Example 3.1 (page 85)

# Responses \( X \sim \text{Binomial}(n, p) \); Observe \( X = r \). Exact
1 - \( \alpha \) lower confidence limit \( p \) satisfies

\[ \alpha = P(X \geq r|p) = \sum_{i=r}^{n} \binom{n}{i} p^i (1-p)^{n-i}. \]

\[ \operatorname{blci} <- \text{function}(r,n,\alpha=0.05,\text{eps}=1e-3) \{ \]
+ if(r <= 0) pl <- 0 else {
+ plu <- \( r/n \)
+ pll <- 0
+ pl <- (pll + plu)/2
+ u <- 1 - \( \text{pbinom}(r - 1, n, pl) \)
+ } else {
+ plu <- pl
+ pl <- (pll + plu)/2
+ u <- 1 - \( \text{pbinom}(r - 1, n, pl) \)
+ }
+ }
+ c(pl=p,neval=i)
+ }

\[ \text{blci}(5,15,\text{eps}=1e-2) \]
\[ \text{blci}(5,15,\text{eps}=1e-3) \]
\[ \text{blci}(5,15,\text{eps}=1e-4) \]
\[ \text{blci}(5,15,\text{eps}=1e-6) \]

(Inverse) Quadratic Interpolation (Brent’s method): Solve \( f(x) = 0 \). Fit quadratic \( q(y) \) to \( \{f(x_i); x_i\} \), \( i = 1; 2; 3 \).

New guess \( x = q(0) \). See NR Section 9.3. Use \texttt{uniroot()} in Splus.

\[ \# \text{ solve } f(p)=0 \text{ for lower binomial confidence limit} \]
\[ f <- \text{function}(p,r,n,\alpha=0.05) \; 1-\text{pbinom}(r-1,n,p)-\alpha \]
\[ \text{unlist(uniroot}(f,c(0,5/15),r=5,n=15, \]
+ tol=1e-4)(c(1:4,6)) \]
\[ \text{unlist(uniroot}(f,c(0.5/15),x=6,n=15,\text{tol}=1e-6)(c(1:4,6))) \]

Newton’s method:

\[ x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})} \]

Rates of Convergence

\( x_1, x_2, \ldots \rightarrow x^* \)

Linear convergence:

\[ |x_{i+1} - x^*| < c|x_i - x^*|, \]

for some \( 0 < c < 1 \), for \( i > I \).

Super linear convergence:

\[ |x_{i+1} - x^*| < c_i|x_i - x^*|^p, \]

for some \( 0 < c < 1 \), for \( i > I, p = 2 \Rightarrow \) quadratic
Bisection: linear with \( c = 1/2 \)

Newton’s method: quadratic

Illinois method (modified secant bracket method): order 1.44

One-Dimensional Minimization

Determine \( x \) such that \( f(x) \leq f(x) \) for all \( x \) (at least locally)

For continuous \( f \), bracketing a local minimum requires 3 points \( a < b < c \), with \( f(b) < \min(f(a), f(c)) \).

Golden Section Search:

Minimize \( f(x) \) over \([a_0, d_0] \):

1. Set \( \alpha = (3 - \sqrt{5})/2 \), \( b_0 = a_0 + \alpha(d_0 - a_0) \), \( c_0 = a_0 + (1 - \alpha)(d_0 - a_0) \). Note \( c_0 - b_0 = \alpha(c_0 - a_0) \), since \( 1 - 2\alpha = \alpha(1 - \alpha) \).

2. If \( f(b_0) < f(c_0) \), set \( a_1 = a_0, d_1 = c_0, b_1 = a_1 + \alpha(d_1 - a_1) \).
   
   If \( f(b_0) > f(c_0) \), set \( a_1 = b_0, d_1 = d_0, b_1 = c_0, e_1 = a_1 + (1 - \alpha)(d_1 - a_1) \).

3. Set \( (a_0, b_0, c_0, d_0) = (a_1, b_1, c_1, d_1) \) and go to 1.

Repeating until convergence.

At every iteration, \( d_1 - a_1 = (1 - \alpha)(d_0 - a_0) \).

Converges to an endpoint if interval does not contain a minimum.

```r
golden <- function(f,brack.int,eps=1e-4,...) { 
  # function to perform a golden section search for 
  # the minimum of a function of 1 variable (arg f).
  # brack.int is the initial bracketing interval 
  # (fca locates a local minimum within this 
  # interval). terminates when relative width of the 
  # bracketing interval is <= eps.
  g <- (3-sqrt(5))/2 
  xl <- min(brack.int) 
  xu <- max(brack.int) 
  tmp <- g*(xu-xl) 
  xmu <- xu-temp 
  xml <- xl+temp 
  fl <- f(xm,...) 
  fu <- f(xmu,...) 
  while(abs(xu-xl)>(1.e-5*abs(xl))*eps) { 
    if (fl<fu) { 
      xu <- xmu 
    } else { 
      xl <- xm 
      xml <- xmu 
      fl <- f(xm,...) 
      xu <- xu-g*(xu-xl) 
      fu <- f(xu,...) 
    } 
  } 
  return(xu) 
}
```

Constrained Optimization

Notation:

The gradient of \( f : \mathbb{R}^n \to \mathbb{R}^n \) is

\[
\nabla f(x) = \left( \frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n} \right)
\]

The Hessian is

\[
\nabla^2 f(x) = \left( \frac{\partial^2 f(x)}{\partial x_j \partial x_i} \right)
\]

For \( G = (g_1(x),\ldots,g_p(x))^T : \mathbb{R}^n \to \mathbb{R}^p \), the Jacobian (\( g \times p \) is

\[
J_G(x) = \left( \frac{\partial g_1(x)}{\partial x_j}, \ldots, \frac{\partial g_p(x)}{\partial x_j} \right)
\]

Note: \( \nabla^2 f(x) = J_G f(x) \) and \( \nabla f(x) = J_J(x) \).
The directional derivative of \( f : \mathbb{R}^p \to \mathbb{R}^l \) at \( x \) in the direction \( d \) is
\[
\lim_{\epsilon \to 0} \frac{f(x + \epsilon d) - f(x)}{\epsilon} = \left. \frac{\partial}{\partial \epsilon} f(x + \epsilon d) \right|_{\epsilon=0} = d^T \nabla f(x).
\]

\( f : \mathbb{R}^p \to \mathbb{R}^l \) is convex on \( A \) if
\[
\lambda f(a) + (1 - \lambda) f(b) \geq f(\lambda a + (1 - \lambda) b)
\]
for all \( a, b \in A \) and \( 0 < \lambda < 1 \).

A smooth \( f \) is convex on \( A \) if \( \nabla^2 f(x) \) is nnd.

**The Problem**

Find \( x^* \) such that \( f(x^*) \leq f(x) \) for all \( x \) in a neighborhood of \( x^* \). (local minimum)

To find a local max of \( f(x) \), find a local min of \( c - f(x) \).

If \( x^* \) is a local min, then \( \nabla f(x^*) = 0 \) and \( \nabla^2 f(x^*) \) is nnd, then \( x^* \) is a local min.

General approach:
1. Given the current point \( x_0 \), choose a direction \( d \) in which to move next.
2. Find a point \( x_1 = x_0 + \lambda d \) such that \( f(x_1) < f(x_0) \).
3. Set \( x_0 = x_1 \), and repeat (until convergence).

**Example 3.2** (Page 94) Logistic regression, binary
# converged after a small number of iterations
b <- c(-1,-1) # initial value
for (i in 1:7) {
  q <- f1(b,r,z)
  b <- b-solve(q$inf,q$score)
  print(c(q$l,b))
}

Error in .Fortran(if(!cmplx) "dqr" else "zqr",:
subroutine dqr: 4 missing
Dumped
> f1(c(-0.2876821,2.444489e+06),r,z)
$loglik:
[1] Inf
$score:
[1] NA NA
$inf:
[,1] [,2]
[1,] NA NA
[2,] NA NA
> # diverged until numerical errors occurred

Backtracking
Instead of full step, consider
\[
x_1(\lambda) = x_0 - \lambda \nabla^2 f(x_0)^{-1} \nabla f(x_0)
\]
Newton-Raphson converges quickly when close, so
1. compute full step (\( \lambda = 1 \)). If \( f(x_1(1)) < f(x_0) \), keep it and repeat.
2. (Backtracking step). If \( f(x_1(\lambda_j)) \geq f(x_0) \), try \( x_1(\lambda_{j+1}) \), \( j = 1, 2, \ldots \), until \( f(x_1(\lambda_j)) < f(x_0) \).

Actually require eg
\[
f(x_1(\lambda)) < f(x_0) + 10^{-4} |x_1(\lambda) - x_0| \nabla f(x_0),
\]
`Step halving`: \( \lambda_j = 1/2^j \).
In Sec. 9.7 of Numerical Recipes uses quadratic extrapolation.

Positive Definite Factorization
Need \( d = -[\nabla^2 f(x_0)]^{-1} \nabla f(x_0) \) to be a descent direction. Directional derivative is
\[
d \nabla f(x_0) = -[\nabla f(x_0)] [\nabla^2 f(x_0)]^{-1} \nabla f(x_0) < 0
\]
Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$.

If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

schol() algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \max(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$.

(x_0)$ is a descent direction for any pd $A$.

If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

**schol()** algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \min(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$.

If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

**schol()** algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \min(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$.

If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

**schol()** algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \min(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$. If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

**schol()** algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \min(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$. If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.

**schol()** algorithm for factoring $A = (a_{ij})$:

1. Specify ratio and fraction.
2. Set $a_1 = \max(a_{ii})$, $a_2 = \min(n_{ii})$, $a_3 = \max(|a_{ii}|)$, $r = \text{ratm}$, $f = \frac{\text{frac}}{2} + 1$.
3. If $a_1 \leq 0$, return $I$.
4. If $a_1 < a_2$ or $a_2 < a_3f^2$, then set $A = A + \alpha I$ and $a_1 = a_1 + \alpha$, where $\alpha = \frac{a_1r^2 - a_2a_3 - a_1}{(1 - r^2)}$.
5. Set $a_4 = a_1f$.
6. Set $a_5 = a_1r^2$.
7. Attempt ordinary Choleski factorization $A = UU^T$. If $a_6 = \sum_{i=1}^{n-1} a_{ii} > a_3$ for all $i$, then return $U$.
8. If not, set $a_6 = 2a_5$, $A = A + a_4 I$, $a_5 = a_1 + a_4$, and go back to step 6.

Performance is much more sensitive to factor than ratio. Could calculate largest and smallest eigenvalues, instead of diag of $A^{-1}V^f(x_0)$, which is a descent direction for any pd $A$. If $\nabla^f(x_0)$ is not pd, use some $A$ that is, e.g., $\nabla^f(x_0) + \alpha I$.
Aside: Linking FORTRAN to Splus

In the unix shell:

```
% f77 -O -c dschol.f
% ld -B dynamic -G -o S.so dschol.o
```

(can include multiple routines and object files)

In Splus:

```
> dyn.load.shared('S.so') # Splus 3.4
> dyn.open('S.so') # Splus 6.0
> dyn.load('R.so') # R
```

(Automated by the `.First.lib()` function in 3.4)

Alternately, use Splus `CHAPTER` to generate makefile, and run `% Splus make`.

In 6.0 the `library()` command automatically opens the `.so` file.

---

### S Function `nr()`

```r
nr <- function(b, fn, fhn, gtol=1e-5, ptol=1e-8, 
    iter=30, frac=.005, ratm=.0001, stepf=.5, 
    eif=NULL, ...) {
    # Modified Newton's method for minimization
    # Arguments
    # b=initial parameter values
    # fn=function to evaluate function being minimized, 
    # called as fn(b, ...)
    # fhn=function to calc function, gradient, and 
    # hessian of function being minimized, called as 
    # fhn(b, ...)
    # gtol= convergence if max(abs(gradient)) < gtol
    # ptol= convergence if rel change in b < ptol
    # iter=max # iterations (input), 
    # frac: the amount added to the diagonal of the 
    # hessian when it is not pd is roughly 
    # 2*frac*max(abs(hessian))
    # ratm: the minimum value allowed for any diagonal 
    # element of the factored hessian is roughly 
    # ratm*sqrt(max(abs(hessian)))
    # stepf=the fraction by which the step size is 
    # reduced in each step of the backtracking algorithm
    # eif=if given, eigenvalues of the hessian are 
    # computed when the Choleski factorization fails, 
    # and a constant added to the diagonal to make 
    # the smallest eigenvalue=eif*largest 
    # ... additional arguments to fn and fhn
    # returns a list with components 'b'=minimizing 
    # parameter values, 'value', 'score' and 'hessian' 
    # giving the value of these quantities at b, and 
    # 'comp'=a vector with components 'iter', 
    # giving the number of iterations used, an error 
    # code ('error'=0 no errors, =1 if error in 
    # directional search, =2 if max iterations exceeded), 
    # 'notpd' giving the number of iterations where 
    # the hessian was not pd, and 'steph' giving the 
    # number of times the step length was reduced 
    # < length(b)
    error <- stepf <- notpd <- 0
    for (ll in 1:iter) {
        z <- fhn(b, ...)
        if (max(abs(z[[2]])) < gtol) return(list(b=b, 
            value=z[[1]], score=z[[2]], hessian=z[[3]], 
            comp=c(iter=ll, error=0, notpd=notpd, 
                steph=stepf))
        # if true, iteration converged
        if (is.null(eif)) {
            hc <- schol(z[[3]], frac=frac, ratm=ratm)
            ut <- attr(hc, 'info')
            if (ut>0) stop('factorization failed')
            if (ut<0 | attr(hc, 'dinc') > 0) 
                notpd <- notpd+1
        } else {
            h <- eigen(h, symmetric=TRUE)
            hc <- .Fortran('ochol', as.double(h), 
                as.integer(n), as.double(0), 
                integer(1))[c(1,4)]
            if (hc[[2]]==0) hc <- matrix(hc[[1]], n)
            # could still give a nearly singular matrix--
            # could also check condition number
            else {
                h <- z[[3]]
                hc <- .Fortran('schol', as.double(h), 
                    as.integer(n), as.double(0), 
                    integer(1))[c(1,4)]
                if (hc[[2]]==0) hc <- matrix(hc[[1]], n)
                # could still give a nearly singular matrix--
                # could also check condition number
                else {
                    # if not pd, add a constant to diag(h)
                    notpd <- notpd+1
                    p <- eigen(h, symmetric=TRUE, 
                        only.values=TRUE)$val
                    ```
Example: Weibull Regression (Page 106)

\[
S(t) = \exp(-t^{\delta_1} \exp(\delta_2 + \beta z))
\]

\[
f(\theta) = -\ell(\theta) = -\sum_i \left( \delta_1 \log(S(t_i,z_i)) + \theta_0 \log(v_i) \right)
\]

\[
v_i = \log(S(t_i,z_i)), \theta = (\alpha, \delta, \beta)' \text{. Since} \quad \partial\ell/\partial \alpha = v_i \log(v_i), \partial\ell/\partial \delta = -v_i \exp(\alpha), \text{and} \quad \partial\ell/\partial \beta = z_i \partial\ell/\partial \alpha,
\]

\[
\partial f(\theta)/\partial \alpha = -\sum_i (\delta_1 + (\delta_1 - v_i) \log(v_i))
\]

\[
\partial f(\theta)/\partial \delta_1 = \exp(\alpha) \sum_i (\delta_1 - v_i)
\]

\[
\partial^2 f(\theta)/\partial \alpha^2 = -\sum_i (\delta_1 - v_i) \log(v_i) - v_i \log(v_i) - \sum_i z_i (\delta_1 - v_i)
\]

\[
\partial^2 f(\theta)/\partial \alpha \partial \delta_1 = \exp(\alpha) \sum_i z_i (\delta_1 - v_i)
\]

\[
\partial^2 f(\theta)/\partial \delta_1^2 = \exp(2\alpha) \sum_i v_i
\]

\[
\partial^3 f(\theta)/\partial \delta_1 \partial \alpha = \exp(2\alpha) \sum_i z_i v_i
\]

\[
\partial^3 f(\theta)/\partial \delta_1^2 \partial \beta = \exp(2\alpha) \sum_i z_i^2 v_i\n\]
+ h <- matrix(0,3,3)  
+ h[1,1] <- -sum((fi-v+v*logv)*logv)  
+ h[2,1] <- h[1,2] <- -sum((-fi+v+v*logv))*t4  
+ h[3,1] <- h[1,3] <- -sum(z*(-fi+v+v*logv))*t4  
+ h[2,2] <- sum(v)*t6  
+ h[3,2] <- h[2,3] <- sum(z*v)*t6  
+ h[3,3] <- sum(z^2*v)*t6  
+ list(f,s,h)  
+ }  

> survReg(Surv(ti,fi)~z,d,dist = "weibull")  
Call:  
survReg(formula = Surv(ti, fi) ~ z, data = d,  
dist = "weibull")  
Coefficients:  
(Intercept) z  
2.308686 0.02375106  
Scale= 0.3575803  
Loglik(model)=-477.7 Loglik(intercept only)=-478.1  
Chisq= 0.75 on 1 degrees of freedom, p= 0.39  
n= 200  
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.0005,  
+ time=d$ti,fi=d$fi,z=d$z))  
[1] 0.12 0.01 0.19 0.00 0.00  
> u$comp  
iter error notpd steph  
7 0 2 4  
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.00001,  
+ time=d$ti,fi=d$fi,z=d$z))  
[1] 0.11 0.00 0.11 0.00 0.00  
> u$comp  
iter error notpd steph  
7 0 2 7  
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.00005,  
+ time=d$ti,fi=d$fi,z=d$z))  
[1] 0.10 0.00 0.10 0.00 0.00  
> u$comp  
iter error notpd steph  
16 1 8 7  
Model trust region:  
Region where quadratic approx. should be reliable  
Perform minimization within the model trust region  
If new point is not better, shrink region  
nlminb(): Uses model trust region  
Iteration stops if rel. change in x  
< .Machine$double.eps^(1/2) or rel. change in f  
< 1e-10.  
> wgh <- function(b,time,fi,z){#gradient and hessian
+ # assign('ng',ng+1,frame=0)
+ t4 <- exp(b[1])
+ t6 <- t4*t4
+ v <- (time/exp(b[2]+b[3]*z))^t4,
+ logv <- log(v)
+ s <- -c(sum(fi+(fi-v)*logv),sum((v-fi))*t4,
+ sum(z*(v-fi))*t4)
+ h <- c(-sum((fi-v-v*logv)*logv),-sum((-fi+
+ v+v*logv))*t4,sum(v)*t6,-sum(z*(-fi+v+v*
+ logv))*t4,sum(z*v)*t6,sum(z^2*v)*t6)
+ list(gradient=s,hessian=h)
+}
+
+ unix.time(u <- nlminb(c(0,0,0),fw,gradient=wgh,
+ hessian=T,time=d$ti,fi=d$fi,z=d$z)[1:8])
+[1] 0.15 0.00 0.19 0.00 0.00
+
+ u
+
+ $parameters:
+ [1] 1.02839444 2.30868513 0.02375108
+
+ $objective:
+ [1] 135.1198
+
+ $message:
+ [1] "RELATIVE FUNCTION CONVERGENCE"
+
+ $grad.norm:
+ [1] 8.633842e-09
+
+ $iterations:
+ [1] 6
+
+ $f.evals:
+ [1] 9
+
+ $g.evals:
+ [1] 7
+
+ $hessian:
+ [,1] [,2] [,3]
+ [1,] 260.66437 -84.04584 16.47120
+ [2,] -84.04584 1329.53871 58.06808
+ [3,] 16.47120 58.06808 1327.63250

Computing Derivatives (Page 112)

Symbolic math
Automatic differentiation
Finite differences

The forward difference approximation to
\( \frac{\partial g}{\partial x} (x) \) is
\[ g(x + \varepsilon \delta) - g(x) \]
and the central difference approximation is
\[ g(x + \varepsilon \delta) - g(x - \varepsilon \delta) \]

Rough rule of thumb:
\( \varepsilon = \sqrt{m} \), or \( |x_j|/\sqrt{m} \).

fdjac <- function(PARAM,G,...,eps=
+ sqrt(.Machine$double.neg.eps)) {
+ # Computes a finite difference approximation to
+ # the Jacobian of G
+ # G is a (possibly vector valued) function
+ # PARAM is the Jacobian point where the Jacobian is approximated
+ # ... = additional arguments to G
+ # eps = step size in finite difference approx.
+ # (scaled by max(abs(PARAM[j]),1))
+ X <- length(PARAM)
+ GO <- G(PARAM,...)
+ JAC <- matrix(0,length(G0),N)
+ for (j in 1:N) {
+ X1 <- PARAM
+ X1[j] <- PARAM[j]+eps*max(abs(PARAM[j]),1)
+ JAC[,j] <- (G(X1,...)-GO)/(X1[j]-PARAM[j])
+ JAC
+ }
+ }

> wg <- function(b,time,fi,z) {# just the gradient
+ t4 <- exp(b[1])
+ v <- (time/exp(b[2]+b[3]*z)) ^ t4
+ logv <- log(v)
+ G0 <- G(PARAM,...)
+ JAC <- matrix(0,length(G0),N)
+ for (j in 1:N) {
+ X1 <- PARAM
+ X1[j] <- PARAM[j]+eps*max(abs(PARAM[j]),1)
+ JAC[,j] <- (G(X1,...)-GO)/(X1[j]-PARAM[j])
+ }
+ }

> rbind(fdjac(c(.1,1.3,-.5),fw,time=d$ti,fi=d$fi,
+ z=d$z),wg(c(.1,1.3,-.5),time=d$ti,fi=d$fi,z=d$z))
+ [1,] 260.66437 -84.04584 16.47120
+ [2,] 8.633842e-09 9
+ $g.evals:
+ [1] 7
+ $hessian:
+ [,1] [,2] [,3]
+ [1,] 260.66437 -84.04584 16.47120
+ [2,] -84.04584 1329.53871 58.06808
+ [3,] 16.47120 58.06808 1327.63250
+ $f.evals:
+ [1] 9
+ $g.evals:
+ [1] 7
+ $message:
+ [1] "RELATIVE FUNCTION CONVERGENCE"
+ $grad.norm:
+ [1] 8.633842e-09
\[ f \leftarrow -\sum(f_i \times (b[1] + \log v)) - v \]
\[ v \leftarrow (\text{time} / \exp(b[2] + b[3] \times z))^t^4 \]
\[ t^6 \leftarrow t^4 \times t^4 \]

\[ \text{wh2} \leftarrow \text{function}(b, \text{time}, f_i, z) \text{ (# - log likelihood, + # gradient, and hessian) } + t^4 \leftarrow \exp(b[1]) \]
\[ t^6 \leftarrow t^4 \times t^4 \]
\[ v \leftarrow (\text{time} / \exp(b[2] + b[3] \times z)) \times t^4 \]
\[ \log v \leftarrow \log(v) \]
\[ + f \leftarrow -\sum(f_i \times (b[1] + \log r)) - v \]
\[ + s \leftarrow \text{wg(b, time, f_i, z)} \]
\[ + h \leftarrow \text{fdjac(b, wg, time, f_i, f_i, z)} \]
\[ + \text{list}(f, s, h) \]

\[ \text{unmix.time}(u \leftarrow \text{nr(c(0,0,0),fw,wh2)}, + \text{time} = \text{d$ti$, f_i = d$fi$, z = d$z$}) \]

\[ |\begin{array}{c}
| 0.33 & 0.00 & 0.33 & 0.00 & 0.00 \\
\end{array}| \]

Slide 37

(b) find better point \( x_1 \) in this direction,
(c) update \( A_0 \) to \( A_1 \).

\[ A_1 = A_0 + \frac{1}{y \times s} y s' - \frac{1}{y \times A_0 y} A_0 s s' A_0, \]
\[ y = \nabla f(x_1) - \nabla f(x_0), x = x_1 - x_0. \]

Initial \( A_0 \): usually diagonal; scaling important, can sometimes use \( \max(|f(x_0)|, 1) \) (rescale covariates)

Step size: Try \( x_1 = x_0 + d \), and backtrack if not better. (Can also use a model trust region approach.)

Skip update of \( A \) if \( s'y \) is not large enough (error in old dfmain).

\[ \text{bfgs} \leftarrow \text{function}(b, \text{fn}, \text{gn}, \text{gtol}=1e-5, \text{ptol}=1e-8, \]
\[ \text{iter}=50, \text{stepf}=5, h, \ldots) \}

# bfgs method for minimization
# b=initial parameter values (input),
# fn=function to calculate objective function,
# called as fn(b, ...), must return a scalar value
# gn=function to calc gradient, called as gn(b, ...)
# gtol= convergence if max(abs(gradient)) < gtol
# ptol= convergence if rel change in b < ptol
# iter= max # iterations (input),
# stepf=the fraction by which the step size is
# reduced in each step of the backtracking
# algorithm
# h = initial approximation to the inverse hessian
# (see below for default)

Slide 39
... additional arguments to fn and gn
# returns a list with components 'b'=minimizing parameter values, 'value'=minimum value of fn,
# and 'comp'=vector with components 'iter' giving number of iterations, 'error' giving an error code (0=no errors, 1=error in directional search, 2=max iterations exceeded), 'steph' giving the number of times the step length was reduced, and 'nskip'=# iterations where h was not updated
n <- length(b)
steph <- nskip <- 0
eps <- .Machine$double.neg.eps
f1 <- fn(b,...)
# initial h (h=approx to the inverse hessian)
if (missing(h)) h <- diag(rep(1/max(abs(f1),1),n))
g1 <- gn(b,...)
score <- -c(h%*%g1)
for (ll in 1:iter) {
  if (max(abs(score))<ptol*(max(abs(b))+1e-8))
    return(list(b=b,value=f1,score=g1,
hessinv=h,comp=c(iter=ll,error=0,
nskip=nskip,steph=steph)))

bn <- b+score
f2 <- fn(bn,...)
i <- 0
while (is.na(f2) || f2>f1+(1e-4)*sum(score)) {
  i <- i+1
  steph <- steph+1
  sc <- sc*stepf
  bn <- c(b+sc)
f2 <- fn(bn,...)
  if (i>20) return(list(b=b,value=f1,score=g1,
hessinv=h,comp=c(iter=ll,error=1,nskip=nskip,
steph=steph)))
g2 <- gn(bn,...)
  if (max(abs(g2)) < gtol)
    return(list(b=bn,value=f2,score=g2,
hessinv=h,comp=c(iter=ll,error=0,
nskip=nskip,steph=steph)))
  for (ll in 1:iter) {
    if (max(abs(score))<ptol*(max(abs(b))+1e-8))
      return(list(b=b,value=f1,score=g1,
hessinv=h,comp=c(iter=ll,error=1,nskip=nskip,
steph=steph)))
  }
  sc <- score
  if the Fortran routine is not available, replace the lines from here to the next comment with those above
sc <- .Fortran('bfgsup',as.integer(n),as.double(b),as.double(bn),as.double(g1),as.double(g2),as.double(h),as.double(eps),as.integer(nskip),double(n))[[c(6,8:9)]]
sc <- -c(h%*%g2)
  if the Fortran routine is not available, replace the lines from here to the next comment with those above

  sc <- .Fortran('bfgsup',as.integer(n),as.double(b),as.double(bn),as.double(g1),as.double(g2),as.double(h),as.double(eps),as.integer(nskip),double(n))[[c(6,8:9)]]
b <- bn
f1 <- f2
return(list(b=b,value=f2,score=g2,hessinv=h,comp=c(iter=ll,error=2,nskip=nskip,steph=steph)))
}
Using the compiled bfgsup is substantially faster.
c inputs are n, b, bn, g1, g2, h, eps, and nskip, c as defined in bfgs(). outputs are the updated h, c the new search direction h x=g2, and nskip c (updated if the update of h is skipped)
subroutine bfgsup(n,b,bn,g1,g2,h,eps,nskip,sc)
double precision b(n),bn(n),g1(n),g2(n),h(n,n),
$ eps,sc(n)$ integer n,nskip,i,j
double precision t1,t2,t3,t4,dot
do 5 i=1,n
  g1(i)=g2(i)-g1(i)
  b(i)=ddot(b,n,1,h(i,i),1)
end do
continue
do 10 i=1,n
  bn(i)=bn(i)-b(i)
b(i)=ddot(b,n,g1,1,h(i,i),1)
end do
continue
\[ t_1 = \ddot{n}(g_1, 1, b_1) \]
\[ t_2 = \ddot{n}(g_1, 1, b, 1) \]
\[ t_3 = \ddot{n}(g_1, 1, g_1, 1) \]
\[ t_4 = \ddot{n}(b_1, 1, b_1, 1) \]

if \( (t_1 \leq 0 \text{ or } t_1 \cdot t_1 \leq \varepsilon_3 \cdot t_3 \cdot t_4) \) then

\[ nskip = nskip + 1 \]
goto 50
endif

\[ t_3 = \sqrt{t_1} \]
\[ t_2 = \sqrt{t_2} \]
\[ t_1 = t_2 / t_3 \]

do 15 i = 1, n

\[ b_n(i) = b_n(i) / t_3 \]
\[ b(i) = b(i) / t_2 \]
\[ g_1(i) = b_n(i) \cdot t_1 - b(i) \]

15 continue

do 20 i = 1, n

do 21 j = i, n

\[ h(j, i) = h(j, i) + b_n(i) \cdot b_n(j) - b(i) \cdot b(j) + \]
\[ g_1(i) \cdot g_1(j) \]

21 continue

20 continue

50 do 30 i = 1, n

\[ s(i) = -\ddot{n}(g_2, 1, h(1, i), 1) \]

30 continue

return
end

Below \( f_v() \) and \( g_v() \) are the same as before.

> unix.time(u <- bfgs(c(0,0,0),fw,vg,
+ time=d$ti,fi=d$f1,z=d$z))
[1] 0.12 0.00 0.12 0.00 0.00
> u

$\textbf{b}$:
[1] 1.0283944 2.3086851 0.0237511
$\textbf{value}$:
[1] 135.1198
$\textbf{score}$:
[1] -1.011132e-06 -2.125407e-07 5.464966e-06
$\textbf{hessinv}$:
[,1] [,2] [,3]
[1,] 3.890873e-03 2.391629e-04 -4.587204e-05
[2,] 2.391629e-04 7.602915e-04 -4.194658e-05
[3,] -4.587204e-05 -4.194658e-05 7.274173e-04
$\textbf{comp}$:
iter  error  nskip  steph
1  5009

Nearly as efficient as \( \texttt{nr()} \)
Finite difference approximation to gradient:

> unix.time(u2 <- bfgs(c(0,0,0),fw,
+ function(b,...) c(fdjac(b,fw,...)),
+ time=d$ti,fi=d$f1,z=d$z))
[1] 0.48 0.00 0.48 0.00 0.00
> u2

$\textbf{b}$:
[1] 1.0283944 2.3086851 0.0237511
$\textbf{value}$:
[1] 135.1198
$\textbf{score}$:
[1] 5.245844e-06 -2.336740e-06 1.078959e-05
$\textbf{hessinv}$:
[,1] [,2] [,3]
[1,] 262.18042 -81.82152 11.81525
[2,] -81.82152 1345.01779 72.40078
[3,] 11.81525 72.40078 1379.64689
$\textbf{comp}$:
iter  error  nskip  steph
1  700

gtol criterion not met, but converged by pto1. Errors in finite difference would cause problems without pto1.

BFGS in \( \texttt{nlminb()} \):

> unix.time(u <- nlminb(c(0,0,0),fw,gradient=wg,
+ time=d$ti,fi=d$f1,z=d$z))
[1] 0.26 0.00 0.26 0.00 0.00
> u
Quasi-Newton methods based on rank 1 updates can be developed (page 124, notes, and Lange, 1999, Section 11.6).

The Nelder-Mead Simplex Method (Page 125)

- Start with $p+1$ points (a simplex) in $R^p$
- Reflect the worst point ($P_1$) through the center of the others ($C$) giving $P_4 = P_1 + 2(C - P_1)$
  - If $P_4$ is within the range of the others, keep it
  - If $P_4$ is best, try a longer step ($P_5 = P_4 + C - P_1$)
  - If $P_4$ is still worst,
    - If $P_4$ is better than $P_1$, try a shorter step ($P_6 = P_4 - (C - P_1)/2$)
    - Otherwise try $P_7 = (P_1 + C)/2$
- Continue until function values are all within some tolerance
- Useful if derivatives difficult to calculate or if function evaluations have significant error.
- Slow linear rate near solution
- Can collapse into a lower dimensional subspace

```r
simplex <- function(b, fun, del = 1, ftol = 1e-8, itmax = 1000, ...) {
  # minimization using the Nelder-Mead simplex
  # method, based on num-rec routine amoeba
  # output gives the minimizing parameter value ($b$),
  # the minimum value of fun ($value$), and `$comp`
  # giving the number of function evaluations and an
  # error code (1=did not converge)
  # Input:
  # b=initial values, fun=function to be minimized,
  # called as fn(b,...), del=the step used from b in
  # each coordinate direction to set up the initial
  # simplex, ftol=convergence criterion--max relative
  # difference between highest and lowest point
  iter <- 0
  np <- length(b)
  p <- rep(b, np)
  dim(p) <- c(np, np)
  p <- t(p)
  diag(p) <- diag(p) + del
  p <- rbind(b, p)
  y <- rep(0, np+1)
  for (i in 1:(np+1)) y[i] <- fun(p[i,],...)
  # now <- apply(p, 2, sum)
  while (iter < itmax) {
    o <- order(y) # don't need a full sort,
    p <- p[o] # only smallest and two largest,
    for (j in 1:3) {
      if (y[o[j]] > y[o[j+1]])
        if (y[o[j]] < y[o[j+1]])
    }
  }
  return(list(b = p[1], value = y[1], comp = 1))
}
y <- y[0] # so could be done more efficiently
ilo <- 1
ihi <- np+1

rtol <- 2*(abs(y[ihi]-y[ilo])/
(abs(y[ihi])+abs(y[ilo])+1e-8))
if (rtol < ftol) return(list(b=p[ilo,],
    value=y[ilo],comp=c(iter=iter,error=0)))
if (iter >= itmax) return(list(b=p[ilo,],
    value=y[ilo],comp=c(iter=iter,error=1)))

iter <- iter+2
# new point chosen by reflecting the worst current
# through the plane of the others
psum <- psum-p[ihi,] # c-p1
p4 <- p[ihi,]+2*bstep
y4 <- fun(p4,...)
if (y4 <= y[ilo]) { # new point is best--
p5 <- p4+bstep # try going further
y5 <- fun(p5,...)
if (y5<y4) {
y[ihi]<- y5; psum<- psum+p5; p[ihi,]<- p5
} else {
    y[ihi]<- y4; psum<- psum+p4; p[ihi,]<- p4
} } else if (y4 >= y[inhi]) {
    if (y4< y[ihi]) p6 <- p[ihi,]+bstep/2 else
    p6 <- p[ihi,]+bstep/2
    y6 <- fun(p6,...)
    if (y6>y[ihi]) {
        y[ihi]< y6; psum<- psum+p6; p[ihi,]<- p6
    } else { # still bad, shrink simplex
        for (i in (1:(np+1))[-ilo]) {
            psum <- (p[i,]+p[ilo,])/2
            p[i,] <- psum
            y[i] <- fun(psum,...)
        }
        iter <- iter+np
        psum <- apply(p,2,sum)
    }
    if (y[ihi]< y[inhi])
        y[ihi]<- y6; psum<- psum+p6; p[ihi,]<- p6
    } else {
        y[ihi]<- y4; psum<- psum+p4; p[ihi,]<- p4
        iter <- iter-1
    }
}

> # Weibull example
> unix.time(u <- simplex(c(0,0,0),fw,
+    time=d$ti,fi=d$fi,z=d$z))
[1] 0.40 0.00 0.43 0.00 0.00
> u

$\$: b:
[1] 1.02834543 2.30864089 0.02376035
$\$: value:
[1] 135.1198
$\$: comp:
iter error
132 0
> unix.time(u <- simplex(c(0,0,0),fw,ftol=1e-12,
+    time=d$ti,fi=d$fi,z=d$z))
[1] 0.49 0.00 0.50 0.00 0.00
> u
$b:
[1] 1.02839421 2.30868513 0.02375152
$\$: value:
[1] 135.1198

> # comp:
> iter error
> 176 0

Algorithm has linear rate of convergence.

A Neural Network Classification Model (page 128)

Goal: classify objects into $K$ classes based on $p$-dimensional vector $x$.

Training data: $x_i = (x_{i1}, \ldots, x_{ip})$, $y_i = k$ if object $i$ is in
class $k$, $i = 1, \ldots, m$.

Use training data to construct a rule for classifying new cases from observed $x$.

Medical diagnosis: Classes = possible diagnoses, $x =$ patient’s symptoms and history, lab test results, etc.

Fisher’s iris data: Classes = 3 species of iris, $(x_1, \ldots, x_4)$ = sepal length, sepal width, petal length, and petal width.

Classification methods: discriminant analysis, (polynomial) logistic regression, nearest neighbor methods, flexible discriminant analysis, penalized discriminant analysis (see Hastie, Tibshirani and Buja, 1994), many others.

Optimal Bayes rule (equal costs): Assign to class $k$ if
Can also replace

\[ P(y = k|x) > P(y = j|x) \text{ for all } j \neq k. \]

**Feed-Forward Neural Network**

Input layer \( (x_{ij}, j = 1, \ldots, p) \), hidden layer(s) of nodes, output layer \((K \text{ outputs})\).

Each node in each layer passes an output to every node in the next layer. At each node, inputs combined with weights to get output value. Outputs for each layer transformed to a common scale, eg with \( g(z) = \frac{1}{1 + \exp(-z)} \).

Transformation of the final output sometimes different (linear, threshold).

Can uniformly approximate smooth functions on a compact set using a single hidden layer with \( N \) nodes.

For classification, if use \( K \) output nodes for the \( K \) class probabilities, and a single hidden layer with \( N \) nodes, then get

\[ P(y_i = k|x_i) = p(k; x_i, v, w) = g \left( v_{0k} + \sum_{n=1}^{N} v_{nk} g(z) \right). \]

Weights \( w_{jk} \) and \( v_{nk} \) are unknown parameters
\( (N(p + 1) + K(N + 1) \text{ parameters}; \ p = 4, K = 3, N = 5 \Rightarrow 43 \text{ parameters}) \).

Can also replace \( v_{0k} \) by \( x^T 3k \).

Generally \( \sum_{k} P(y = k|x) \neq 1 \), but the probabilities can be renormalized.

**Backpropagation algorithm for training:**

**Objective function**
\[ \sum_{i} \sum_{k} \{ I(y_i = k) - p(k|x_i, v, w) \}^2 \]

Feed training set data into the network to compute \( p \)'s at current \( v, w \).

Send \( v, p \) back through the network, updating the weights at each node.

Can compute components of gradient of objective function using only local information. Shift weights a small amount in direction of negative gradient (ie, steepest descent).

Neural networks have many parameters, and can overfit the training data. When overfit, \( p(y|x, v, w) \) is not very smooth in \( x \), so near neighbors in \( x \) can have very different \( p \)'s. Such a classifier is unlikely to perform well on new data. Another description:

Overassociating, overextending, creating infinitesimal, worthless categories in which everything belonged always and only to itself.

* Powers, 1995.*

**Standard paradigm:**
* Divide data into training and test sets.
* Use training data for estimating the weights.
* Estimate classification error from test data.
* Stop updating once classification errors stop decreasing.

Can also use cross validation (need to repeat fitting process many times)

Process does not minimize objective function, but searches for weights that give good predictions in the test data.

Different starting points can lead to quite different weights, and sometimes quite different classifiers.

Neural networks often have many local optima.

Instead, can minimize a penalized objective function
\[ F(v, w) = \sum_{i} \sum_{k} \{ I(y_i = k) - p(k|x_i, v, w) \}^2 \]
\[ + \lambda \sum_{n=1}^{N} \left( \sum_{j=0}^{p} w_{nj}^2 + \sum_{k=1}^{K} v_{nk}^2 \right) \]

(v_{nj} not penalized). Large \( \lambda \) force smaller weights.

Still can have multiple local minima, so fit many times from different starting points. (Venables and Ripley suggest averaging the probabilities over different local optima.)

Could choose \( \lambda \) to minimize classification errors in a test data set, or in cross validation.

Setting \( r_{ik} = I(y_i = k) - p(k, x_i, v, w) \), \( u_{0n} = g(z_{0n}) \), \( u_{0n} = 1, x_{0n} = 1, \) and \( a_{0k} = v_{0k} + \sum_{n=1}^{N} v_{0n}u_{0n} \), gives

\[
\frac{\partial F(v, w)}{\partial h_k} = -2 \sum_{i=1}^{m} r_{ih}^k g'(a_{ih})u_{0h} + 2\lambda v_{0h}I(h > 0),
\]

\( h = 0, \ldots, N, l = 1, \ldots, K, \)

\[
\frac{\partial F(v, w)}{\partial w_{jk}} = -2 \sum_{i=1}^{m} \sum_{k=1}^{K} r_{ik} g'(a_{ik})u_{0k} g'(z_{0k})x_{ij} + 2\lambda w_{jk},
\]

\( j = 0, \ldots, p \) and \( h = 1, \ldots, N. \)

No reason to calculate Hessian, so use BFGS to minimize \( F(v, w) \) for fixed \( \lambda \).

Slide 61

Functions: nnet() in V&R library; nnfit() in course library.

nnfit() uses random \( U(0,1) \) weights as starting values.

nn.classif() computes estimated probabilities and determines the class with highest estimated probability.

Not a Bayes rule, since have not averaged over priors.

Processing the NN output with eg linear discriminant analysis can give better classification (Hastie, Tibshirani and Buja, 1994, JASA).

\[
gpen \left[ \text{seq}(1,(n+1)*k,by=n+1) \right] \leftarrow 0
\]

\[
\text{unlist.}(\text{Fortran}('nng',\text{as.integer}(\text{nnrow}(x)), \text{as.integer}(p), \text{as.integer}(n), \text{as.integer}(k), \text{as.double(t(x))), \text{as.integer}(y), \text{as.double(b[1:((n+1)*k)]}, \text{as.double(b[(((n+1)*k)+1):length(b)]), obj=double(1),gv=double(n*k+k), gw=double(n*p+n),double(n)))[c(10,11)])+gpen
\]

--- nng.f -------------

\[
gpen[seq(1,(n+1)*k),by=n+1]) \leftarrow 0
\]

\[
\text{unlist.}(\text{Fortran}('nng',\text{as.integer}(\text{nnrow}(x)), \text{as.integer}(p), \text{as.integer}(n), \text{as.integer}(k), \text{as.double(t(x))), \text{as.integer}(y), \text{as.double(b[1:((n+1)*k)]}, \text{as.double(b[(((n+1)*k)+1):length(b)]), obj=double(1),gv=double(n*k+k), gw=double(n*p+n),double(n)))[c(10,11)])+gpen
\]
21 continue do 30 l=1,k
   q1=v(0,l)+ddot(n,v(1,l),1,z,1)
   q1=1/(1+exp(-q1))
   if (iy(i).eq.l) then
     rl=1-ql
   else
     rl=-ql
   endif
   obj=obj+rl*rl
   do 40 j=1,n
     gv(0,j)=gv(0,l)+ql
     qv(j,l)=gv(j,l)+ql*z(j)
     t1=qv(j,l)*z(j)*1-z(j))
     do 42 j2=1,np
       gv(j2,j)=gv(j2,j)+t1*x(j2,i)
     42 continue
   obj=double(1),double(n)$obj+pen
   40 continue
  30 continue
  20 continue

Slide 65

Slide 66

Slide 67

Slide 68
lambda=lambda)
list(weights=list(v=matrix(u[[1]][1:n2],
nrow=N+1),w=matrix(u[[1]][(n2+1):nwt],
nrow=p+1)),dims=c(p=p,K=K,N=N,m=m),
message=NULL,covtran=rr,perf=c(u$comp,
score=sqrt(sum(u$score*u$score))))
} else {
u<- nlminb(init,nnf,nng,control=nlminb.control(
maxiter=maxiter),y=y,x=x,lambda=lambda)
list(weights=list(v=matrix(u[[1]][1:n2],nrow=
N+1),w=matrix(u[[1]][(n2+1):nwt],nrow=
p+1)),dims=c(p=p,K=K,N=N,m=m),message=
u[[3]],covtran=rr,perf=unlist(u[c(2,4:7)]))
}
---------------------------------------------------

nnp <- function(nnobj,x) {
  # computes the estimated class probabilities for a
  # neural network classification model fit
  # nnobj=output from nnfit()
  # x=matrix of feature vectors where probabilities
  # are to be calculated
  # output = K x nrow(x) matrix whose ith column
  # gives the estimated probabilities that the
  # feature vector in the ith row of x is in
  # each of the K classes
  x <- as.matrix(x)
p <- nnobj$dims[1]
if (ncol(x) != p) stop('wrong # covs')
for (i in 1:p) {
x[i,i] <- (x[i,]-nnobj$covtran[i,1])/
(nnobj$covtran[i,2]-nnobj$covtran[i,1])
}
k <- nnobj$dims[2]
s <- nnobj$dims[3]
l1 <- (n+1)*k
nwt <- length(nnobj$weights)
probs <- .Fortran('nnp',as.integer(nrow(x)),
as.integer(p),as.integer(n),
as.integer(k),as.double(t(x)),
as.double(nnobj$weights$v),
as.double(nnobj$weights$w),
probs=double(k*nrow(x)),double(n))$probs
dim(probs) <- c(k,nrow(x))
probs
}
---------------------------------------------------

nn.classif <- function(nnobj,xx) {
  # determine class with highest estimated class
  # probability (there may be better classifiers)
  ap <- nnp(nnobj,xx)
  u <- apply(ap,2,function(u) (1:length(u))[u ==
max(u)][1])
  list(probs=ap,predclass=u)
}
Example 3.3 Fisher’s Iris Data
Iris data:
iris
included with Splus.
features: sepal length, sepal width, petal length, and
petal width
classes: species (Setosa, Versicolor, Virginica).
Training set: first 33 obs from each species (xx1 and yy1)
Test data: other 17 obs from each species (xx2 and yy2).
```r
> dim(iris)
[1] 50 4 3
> xx1 <- rbind(iris[1:33,,1],iris[1:33,,2],
+ iris[1:33,,3])
> xx2 <- rbind(iris[34:50,,1],iris[34:50,,2],
+ iris[34:50,,3])
> dim(xx1)
[1] 99 4
> yy1 <- c(rep(1,33),rep(2,33),rep(3,33))
> yy2 <- c(rep(1,17),rep(2,17),rep(3,17))
> a <- nnfit(yy1,xx1,lambda=.1,N=2)
> names(a)
[1] "weights" "dims" "message" "covtran"
[5] "perf"
> a$perf
          iter  error  nskip steph  score
163     0 1.187332e-05
> a$dims
  p K H m
4 3 2 99
```

Classification error in the training set (left side) and test set (right side):

Slide 73

```r
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),
+ table(tmp1$predclass[100:150],yy2))
     1 2 3
1 300 170 0
2 100 0 61
3 300 20 6
Decreasing \( \lambda \) gives a perfect fit for the training data, but not for the test data:

Slide 74

```r
> a <- nnfit(yy1,xx1,lambda=.001,N=2)
> a$perf
          iter  error  nskip steph  score
311     0 1.288065e-05
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),
+ table(tmp1$predclass[100:150],yy2))
     1 2 3
1 300 170 0
2 300 0 61
3 300 20 6
Suggests \( \lambda = .1 \) might be reasonable. \( \lambda = 0 \Rightarrow \) multiple local optima:

Slide 75

```r
> a <- nnfit(yy1,xx1,lambda=0,N=2)
> a$perf
          iter  error  nskip steph  score
2000 1.956408e-09
> tmp2 <- nn.classif(a2,rbind(xx1,xx2))
> cbind(table(tmp2$predclass[1:99],yy1),
+ table(tmp2$predclass[100:150],yy2))
     1 2 3
1 300 170 0
2 300 0 61
3 300 20 6
> summary(tmp1$probs-tmp2$probs)
                Min. 1st Qu. Median Mean
[,1] -4.5343606 7.233864 -7.8343751
[,2] -0.9652276 -15.235744 16.6803599
[,3] 9.0293324 -11.777118 -0.2712737
```

Increasing \( \lambda \):

Slide 76
Example 3.4: Simulated Data

Example 3.4: Simulated Data (Page 141) 3 classes, 4 covariates

\[
\begin{align*}
\text{Slide 77} \\
\text{Summary of Output} \\
\text{% Correctly Classified} \\
\hline
\text{lambda} & \# \text{ iters} & \text{Obj} & \text{Train} & \text{Test} \\
\text{10} & 50 & 323 & 43 & 43 \\
\text{1} & 75 & 319 & 50 & 46 \\
\text{.1} & 157 & 293 & 55 & 50 \\
\text{.01} & 692 & 261 & 67 & 49 \\
\text{.001} & 498 & 232 & 71 & 45 \\
\text{.0001} & 727 & 220 & 71 & 49 \\
\text{.00001} & 1556 & 221 & 70 & 42 \\
\text{.000001} & 2531 & 208 & 76 & 43 \\
\text{.0001(2)} & 1041 & 218 & 73 & 44 \\
\end{align*}
\]

3rd Qu. Max.

\[
\begin{align*}
-0.3638123 & \quad 0.6102857 & \quad 0.7461173 & \quad 0.4990865 \\
\end{align*}
\]

Solving Nonlinear Equations

Newton’s Method

\[
G(x) = 0.
\]

Repeatedly solve tangent plane approximation

\[
G(x) \approx G(x_0) + J(x_0)(x - x_0),
\]

giving

\[
x_1 = x_0 - J(x_0)^{-1}G(x_0),
\]

where

\[
J(x) = \left( \frac{\partial g_i(x)}{\partial x_j} \right).
\]

Often fails, and no objective function. Can monitor value of

\[
f(x) = G(x)^\top G(x)/2 = \sum_k g_k(x)^2/2.
\]

Since

\[
\frac{\partial f(x)}{\partial x_j} = \sum_k g_k(x) \frac{\partial g_k(x)}{\partial x_j} = \left( \frac{\partial G(x)}{\partial x_j} \right)^\top G(x),
\]

\[
\nabla f(x) = J(x)^\top G(x). \quad \text{Thus the directional derivative of } f \text{ in the Newton direction } d = -J(x_0)^{-1}G(x_0) \text{ is}
\]

\[
d^\top \nabla f(x_0) = -G(x_0)^\top [J(x_0)^{-1}]^\top J(x_0)G(x_0)
\]

\[
= -G(x_0)^\top G(x_0) < 0.
\]

Try full step, and backtrack if it does not improve \( f \).
**BIO 248, printed February 24, 2002**

Complete data scores \( \sum_i R_i U(y_i, x_i, z_i, \theta) \), where

\[
U(y, x, z, \theta) = \begin{pmatrix} 1 \\ x' \end{pmatrix} \left[ y - p(x, z, \theta) \right].
\]

and \( p(x, z, \theta) = P(y_i = 1|x_i, z_i) \), are unbiased.

\[
\sum_i R_i U(y_i, x_i, z_i, \theta) + \left( 1 - \frac{R_i}{\pi} \right) \phi(y_i, x_i, \theta) = 0
\]

is also unbiased for \( \theta \). For any \( n \)-vector function \( \phi \), choose \( \phi \) to minimize the variance of \( \theta \). Optimal choice (RRZ, 1994)

\[
\phi^*(y, x, \theta) = E_y[x_i|y, x_i]U(y_i, x_i, z_i, \theta).
\]

An approximation:

\[
\phi(y, x, \theta) = U(y_i, x_i, E[z_i|y_i, x_i], \theta).
\]

Estimate \( E[z_i|y_i, x_i] \) with fitted values \( z^*_i \) from

\[ z^*_i = \mu_0 + \mu_1 y_i + \mu_2 x_i + \epsilon_i \]

(fit using only complete cases, but need \( z^*_i \) for all \( i \)).

**BIO 248, printed February 24, 2002**

Complete data scores \( \sum_i R_i U(y_i, x_i, z_i, \theta) \), where

\[
U(y, x, z, \theta) = \begin{pmatrix} 1 \\ x' \end{pmatrix} \left[ y - p(x, z, \theta) \right].
\]

and \( p(x, z, \theta) = P(y_i = 1|x_i, z_i) \), are unbiased.

\[
\sum_i R_i U(y_i, x_i, z_i, \theta) + \left( 1 - \frac{R_i}{\pi} \right) \phi(y_i, x_i, \theta) = 0
\]

is also unbiased for \( \theta \). For any \( n \)-vector function \( \phi \), choose \( \phi \) to minimize the variance of \( \theta \). Optimal choice (RRZ, 1994)

\[
\phi^*(y, x, \theta) = E_y[x_i|y, x_i]U(y_i, x_i, z_i, \theta).
\]

An approximation:

\[
\phi(y, x, \theta) = U(y_i, x_i, E[z_i|y_i, x_i], \theta).
\]

Estimate \( E[z_i|y_i, x_i] \) with fitted values \( z^*_i \) from

\[ z^*_i = \mu_0 + \mu_1 y_i + \mu_2 x_i + \epsilon_i \]

(fit using only complete cases, but need \( z^*_i \) for all \( i \)).

**BIO 248, printed February 24, 2002**

Complete data scores \( \sum_i R_i U(y_i, x_i, z_i, \theta) \), where

\[
U(y, x, z, \theta) = \begin{pmatrix} 1 \\ x' \end{pmatrix} \left[ y - p(x, z, \theta) \right].
\]

and \( p(x, z, \theta) = P(y_i = 1|x_i, z_i) \), are unbiased.

\[
\sum_i R_i U(y_i, x_i, z_i, \theta) + \left( 1 - \frac{R_i}{\pi} \right) \phi(y_i, x_i, \theta) = 0
\]

is also unbiased for \( \theta \). For any \( n \)-vector function \( \phi \), choose \( \phi \) to minimize the variance of \( \theta \). Optimal choice (RRZ, 1994)

\[
\phi^*(y, x, \theta) = E_y[x_i|y, x_i]U(y_i, x_i, z_i, \theta).
\]

An approximation:

\[
\phi(y, x, \theta) = U(y_i, x_i, E[z_i|y_i, x_i], \theta).
\]

Estimate \( E[z_i|y_i, x_i] \) with fitted values \( z^*_i \) from

\[ z^*_i = \mu_0 + \mu_1 y_i + \mu_2 x_i + \epsilon_i \]

(fit using only complete cases, but need \( z^*_i \) for all \( i \)).
+ z1 <- cbind(rep(1,n)[!mi],comp[!mi,],miss[!mi,])
+ pr1 <- exp(c(z1 %*% b))
+ pr1 <- pr1/(1+pr1)^2
+ u <- -t(z1)%*%(pr1*z1)/pihat
+ z1 <- cbind(rep(1,n),comp,cov.pred)
+ pr1 <- exp(c(z1 %*% b))
+ pr1 <- pr1/(1+pr1)^2
+ u - t(z1) %*% ((pihat-1+mi)*pr1*z1)/pihat
+
> # generate some data: 4 covs, pairwise cor. rho
> .Random.seed
[1] 17 12 2 22 23 0 55 14 41 4 41 1
> rho <- .5
> n <- 200
> p <- 4
> nmis <- 2 # # covariates with some values missing
> alpha <- -1 #constant term
> beta <- c(1,-1,1,-1) # true regression coef
> miss.prob <- .5 # prob z_ij missing for each i
> a <- matrix(rho,p,p)
> diag(a) <- 1
> Z <- rmnorm(n,p,cov=a) #course library fcn
> cor(t(Z))
[1,] 1.0000000 0.5058296 0.5109325 0.4887334
[2,] 0.5058296 1.0000000 0.5157523 0.5055901
[3,] 0.5109325 0.5157523 0.9999999 0.4875875
[4,] 0.4887334 0.5055901 0.4875875 1.0000000
> rp <- exp(alpha+c(beta%*%Z))
> rp <- rp/(1+rp)
> resp <- ifelse(runif(n)<rp,1,0)
> comp <- t(Z[1:(p-nmis),])
> miss <- t(Z[(p-nmis+1):p,])
> miss[runif(nmis*n)<miss.prob] <- NA
> # identify missing values
> mi <- is.na(apply(miss,1,sum))
> table(mi)
      FALSE TRUE
    42    158
> # regression to estimate conditional expectations
> newd <- data.frame(comp,resp)
> names(newd) <- c('comp1','comp2','resp')
> cov.pred <- predict(lm(miss~comp+resp,subset=!mi),
+ newdata=newd)
> # (computed predicted values for all data points)
> # estimate logistic parameters
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)

$ b:
$f:
[1] 2.396976e-24
$ comp:
iter error step
7 0 0
> # different starting value
> neq(mnorm(p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)
$b:
$f:
[1] 2.122109e-13
$ comp:
iter error step
6 0 0
> # finite difference approximation to Jacobian
> mlrfjac <- function(b,resp,comp,miss,cov.pred,mi)
+ fdjac(b,mlrsc,resp,comp,miss,cov.pred,mi)
> mlrsc(c(alpha,beta),resp,comp,miss,cov.pred,mi)
+ mlrsc(c(alpha,beta),resp,comp,miss,cov.pred,mi)
> mlrjac(c(alpha,beta),resp,comp,miss,cov.pred,mi)

$ b:
$f:
[1] 2.396976e-24
$ comp:
iter error step
7 0 0
> # different starting value
> neq(mnorm(p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)
}$b:
[1] -2.247057 1.667337 -1.878995 2.107479 2.697571
$f:
[1] 5.119976e-15
$ comp:
iter error step
6 0 0
> # different starting value
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)

$ b:
[1] -2.224706 -0.513405 -1.833713 -6.846105 6.644404
$f:
$ comp:
iter error step
7 0 0
> # different starting value
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)

$ b:
[1] -34.329205 -0.513405 1.833713 -6.846105 6.644404
$f:
$ comp:
iter error step
7 0 0
> # different starting value
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,
+ miss=miss,cov.pred=cov.pred,mi=mi)
To solve
\[ G(x) = (g_1(x), \ldots, g_p(x))^\top = (0, \ldots, 0)^\top, \]
iterate the following: solve
\[ g_j(x_1, x_2^{(0)}, \ldots, x_p^{(0)}) = 0 \]
for \( x \in x_1^{(1)}, \ldots, x_2^{(1)}, \ldots, x_p^{(1)} \).

Also called backfitting.