6 Numerical Maximization in Statistics

We want to minimize a function f (usually a negative- log-likelihood or related function) over a parameter region which we believe contains at least a sub-region over which the function is locally convex. In large-sample settings, we expect a very sharp peak near which the function behaves like a quadric surface. The calculus-based theory leads to several important remarks for statistical problems.

- Search for parameters with $\nabla f(\vartheta) = 0$;
- Newton-Raphson (NR) gives one-step solution in case f is quadratic;
- Newton-Raphson converges quadratically, i.e. with distances from the local maximizer squaring at each iteration, if we start close enough;
- step-lengths for gradient ascent are essentially arbitrary but may have to be made artifically small in order to avoid overflows and numerical instabilities;
- NR steps may also be wild and numerically unstable away from the immediate neighborhood of a local max.
- at a not-too-large computational cost, it makes sense to avoid unstable steps by searching along the ray provided by either the gradient or the NR increment to ensure that the function-value decreases at each iteration (*reduction of multivariate to univariate search*).

The last suggestion, together with the requirement to approximate gradients and Hessians via finite-difference schemes, is characteristic of *Quasi-Newton methods*.

References for all of these topics: *Numerical Recipes*, plus general books on optimization like Luenberger, *Optimization by Vector Space Methods*, or general numerical-analysis books like the text of Stoer & Bulirsch often used in MAPL 466 or 666.

6.1 *Coding & Splus Functions Related to Newton-Raphson*

The multivariate *Newton-Raphson* (**NR**) method of solving an equation $g(\mathbf{x}) = 0$, where g is a smooth (k-vector-valued) function of a k-dimensional vector variable $\underline{x}$ whose Jacobian matrix

$$J_g(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_k} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_k} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_k} \end{pmatrix}$$

never vanishes, is to write and implement an equation saying that the linear (first-order Taylor series) approximation about $\mathbf{x}$ to the function at an updated variable value $\mathbf{x}'$ is precisely 0, i.e.

$$g(x) + J_g(\mathbf{x})\,(x' - x) = 0 \quad , \quad \text{or} \qquad x' = x - (J_g(\mathbf{x})^{-1})\,g(x)$$

The key application of this idea which we make in computational statistics is, for a fixed dataset $\mathbf{X}$, to

$$g(\vartheta) = g(\vartheta; \mathbf{X}) = -\nabla_\vartheta \, logLik(\vartheta; \mathbf{X})$$

The Newton-Raphson computational algorithm, which we code below in Splus — both from first principles and by using existing standard functions — is to begin with some *initial value* $\mathbf{x}^{(0)}$ and then iteratively for $m = 0, 1, \ldots$, define

$$\mathbf{x}^{(m+1)} = \mathbf{x}^{(m)} - (J_g(\mathbf{x}^{(m)})^{-1})\,g(\mathbf{x}^{(m)})$$

repeatedly until some termination-criterion is met, usually that either m is equal to a fixed large number (like 25) or $\|\mathbf{x}^{(m+1)} - \mathbf{x}^{(m)}\|$ falls below a fixed tolerance (like 10^{-5}). Here is a simple pair of crude Splus functions. The first one, which numerically approximates gradients, is needed only if we do not have a function implementing an analytical formula for the gradient.

```
> Gradmat
function(parvec, infcn, eps = 1e-06)
{
# Function to calculate the difference-quotient approx gradient
#   (matrix) of an arbitrary input (vector) function    infcn
# Now recoded to use central differences !
        dd <- length(parvec)
        aa <- length(infcn(parvec))
        epsmat <- (diag(dd) * eps)/2
        gmat <- array(0, dim = c(aa, dd))
        for(i in 1:dd)
                gmat[, i] <- (infcn(parvec + epsmat[, i]) - infcn(parvec -
                        epsmat[, i]))/eps
        if(aa > 1)
                gmat
        else c(gmat)
}

NRroot
function(inipar, infcn, nmax = 25, stoptol = 1e-05,
        eps = 1e-06, gradfunc = NULL)
{
    assign("Infcn", infcn, frame = 0)
    assign("Eps", eps, frame = 0)
    if(is.null(gradfunc))
        gradfunc <- function(x)
        Gradmat(x, Infcn, Eps)
    ctr <- 0
    newpar <- inipar
    oldpar <- inipar - 1
    while(ctr < nmax & sqrt(sum((newpar - oldpar)^2)) > stoptol) {
        oldpar <- newpar
        newpar <- oldpar - solve(gradfunc(oldpar), infcn(oldpar))
        ctr <- ctr + 1
    }
    list(nstep = ctr, initial = inipar, final = newpar,
        funcval = infcn(newpar))
}
```

Recall that our most frequent statistical objective in using the NR root-finder for the gradient log-likelihood is to minimize the negative log-likelihood function. There is another simple method of numerical optimization called *Steepest Descent* which, although crude, can often serve as an initialization-stage for a NR method which must be coded 'by hand'. The method is simply to move an initial guess $\mathbf{x}$ to an improved one $\mathbf{x}'$ by making a step in the direction of the negative gradient. What advanced-calculus theory tells is that this method improves the objective-function value as long as the step-size is small enough and positive. A simple implementation is as follows.

```
> GradSrch
function(inipar, infcn, step, nmax = 25, stoptol = 1e-05,
        unitfac = F, eps = 1e-06, gradfunc = NULL)
{
# Function to implement Steepest-descent. The   unitfac  condition
#   indicates whether or not the supplied step-length factor(s) multiply
#  the negative gradient itself, or the unit vector in the same direction.
   assign("Infcn", infcn, frame = 0)
   assign("Eps", eps, frame = 0)
   if(is.null(gradfunc))
         gradfunc <- function(x)
         Gradmat(x, Infcn, Eps)
   steps <- if(length(step) > 1) step else rep(step, nmax)
   newpar <- inipar
   oldpar <- newpar - 1
   ctr <- 0
   while(ctr < nmax & sqrt(sum((newpar - oldpar)^2)) > stoptol) {
         ctr <- ctr + 1
         oldpar <- newpar
         newstep <- gradfunc(oldpar)
         newstep <- if(unitfac) newstep/sum(newstep^2) else newstep
         newpar <- oldpar - steps[ctr] * newstep
   }
   list(nstep = ctr, initial = inipar, final = newpar,
         funcval = infcn(newpar))
}
```

One might begin a likelihood maximization with several gradient steps, if there is no particularly good initial guess available for the unknown parameters. (The step-lengths might be chosen optimally within some range at each iteration: we will show how to do this in Splus below.) After the steepest-descent steps are no longer making rapid progress, one might use some automatic but problem-specific criterion to switch over to NR iterations for more rapid final stages of convergence to a minimizer.

There are three relevant Splus functions which, because they are hard-coded in a lower-level language like C, run much faster than these crude functions. They are: *optimize, nlmin*, and *ms*. First, *optimize* is a univariate function-minimizer which requires

(a) that a bounded search-interval be specified, and

(b) (*as far as I can tell*) that the function to be minimized, even though depending nominally on a scalar variable, can make sense of vectorized inputs.

By contrast, the Splus function *nlmin* has the features

(a) that the vector variable of the function to be minimized is completely *un*restricted (otherwise the Splus function to use is *nlminb*);

(b) that an initial guess for the minimizing value must be supplied.

The function *ms* minimizes nonlinear functions of several variables which, like negative-log-likelihoods, are built as sums of identical functions evaluated at a succession of data-values.

For our purposes in this Section, *optimize* is useful as a general way to choose the best step-length at each stage of a gradient or Newton-Raphson search. All three of the standard Splus functions minimize by using variants of the Newton-Raphson algorithm and are very fast for well-behaved functions.

Let us illustrate next both the newly coded and standard functions in the context of maximizing logistic and probit log-likelihoods.

6.1.1 Estimating Simulated Logistic & Probit Regressions

First create logistic- and probit- regression data with (the same set of four independent binary regressors, with coefficients $0.5, 0.4, -0.3, 0.7$ and intercept -2.

```
> bc <- c(0.5,0.4,-0.3,0.7)
> matcov <- matrix(rbinom(800,1,0.5),ncol=4)
> respLgst <- rbinom(200,1,plogis(-2 + c(matcov %*% bc)))
> respPrbt <- rbinom(200,1,pnorm(-2 + c(matcov %*% bc)))
```

Next construct function to calculate both Logistic and Probit log- likelihoods.

```
> binregLik
function(b0, a0, covmat, yresp, dist = plogis)
{
        pvec <- dist(c(covmat %*% b0) + a0)
        sum(log(ifelse(yresp == 1, pvec, 1 - pvec)))
}
> binregLik(bc,-2,matcov,respLgst, dist=plogis)
[1] -88.10565
> binregLik(bc,-2,matcov,respLgst, dist=pnorm)
[1] -92.24571
> binregLik(bc,-2,matcov,respPrbt, dist=pnorm)
[1] -60.06637
> binregLik(bc,-2,matcov,respPrbt, dist=plogis)
[1] -69.00565
```

Now we define the functions we will use in doing Logistic or Probit Regression maximizations. For simplicity, we begin by maximization over only the first two regression parameters, treating the intercept and the other regression, parameters as known.

```
> tempfunc1 <- function(bb)
   -binregLik(c(bb,-.3,.7),-2,matcov,respLgst)
> tempfunc2 <- function(bb)
   -binregLik(c(bb,-.3,.7),-2,matcov,respPrbt, dist=pnorm)
```

```
> c(Gradmat(c(.3,.3),tempfunc1))
[1] -4.324436 -4.730990
> c(Gradmat(c(.2,.6),tempfunc2))
[1] 1.888556 3.973546
```

Consider now the estimation by Steepest Descents (with all steps equal to -0.05 multiplied by the gradient) and Newton-Raphson, as well as the *nlmin* and *glm* functions. First we do the crudest possible Steepest-Descent, then the same thing using the *GradSrch* function above.

```
> btmp <- c(0.2,0.2)
> tempfunc1(c(0.2,0.2))
[1] 90.00409
> for (i in 1:10) { btmp <- btmp - 0.05*
   Gradmat(btmp,tempfunc1)
   cat(round(c(btmp, tempfunc1(btmp)), digits=5),"  \n") }
0.52941 0.55127 88.16112
0.43557 0.46581 88.06424
0.46686 0.50402 88.04838
0.45242 0.49317 88.04627
0.45586 0.49913 88.04594
0.45337 0.49812 88.04589
0.45353 0.49924 88.04588
0.453   0.49928 88.04587
0.4529  0.49955 88.04587
0.45275 0.49963 88.04587
>  unlist(GradSrch(c(0.2,0.2), tempfunc1, 0.05))
 nstep initial1 initial2    final1    final2  funcval
    16      0.2      0.2 0.4525808 0.4998214 88.04587
> unlist(GradSrch(c(0.2,0.2), tempfunc1, 0.05, unitfac=T))
 nstep initial1 initial2    final1    final2 funcval
    25      0.2      0.2 0.3073977 0.3153444 88.7657
```

So we can see in this setting that convergence by steepest descent is achieved but very slowly, and is worse when we take our fixed step-lengths to multiply the unit-vector in the gradient direction. To speed up convergence, we appeal directly to *NRroot*.

```
> unlist(NRroot(c(0.2,0.2), function(bb) t(Gradmat(bb,tempfunc1)) ))
 nstep initial1 initial2    final1    final2     funcval1     funcval2
     4      0.2      0.2 0.4525694 0.4998326 3.836931e-07 5.400125e-07
> tempfunc1(.Last.value[4:5])
[1] 88.04587
```

Now we can see that convergence to the same final point from the same starting-point as steepest-descent is achieved in 4 iteration-steps by NR, with final gradient of the order 10^{-7}. Now let us compute and compare the 4-parameter maximum-likelihood estimates for the probit model on the logistic-regression data, using first *NRroot* and then the **Splus** functions *nlmin*, *ms*, and *glm*.

```
> unlist(NRroot(rep(0,4), function(bb) t(Gradmat(bb, function(uu)
+    -binregLik(uu,-2,matcov,respLgst,dist=pnorm))) ))
 nstep initial1 initial2 initial3 initial4  final1    final2     final3
     5        0        0        0        0 0.63364 0.5637013 -0.1829429
    final4     funcval1      funcval2      funcval3      funcval4
 0.9061298 1.421085e-08 -4.263256e-08 -5.684342e-08 -7.105427e-08

> unlist(nlmin(function(uu) -binregLik(uu,-2,matcov,respLgst,
    dist=pnorm), rep(0,4)) )
                x1                  x2                   x3
 "0.63363989720753" "0.563701077867386" "-0.182940934641386"
                x4 converged                        conv.type
 "0.906129305996438" "TRUE"    "relative function convergence"

> msobj <- ms(~ -rsp*log(pnorm(-2+v1*b1 + v2*b2 + v3*b3 + v4*b4))-
         (1-rsp)*log(1-pnorm(-2+v1*b1 + v2*b2 + v3*b3 + v4*b4)),
         data=data.frame(matrix(cbind(respLgst,matcov), ncol=5,
         dimnames=list(NULL,c("rsp","v1","v2", "v3","v4")))),
         start=list(b1=0,b2=0,b3=0,b4=0), trace=T)
Iteration:  0 ,  1  function calls, F=  147.4891
Parameters:
[1] 0 0 0 0
Iteration:  1 ,  2  function calls, F=  90.02893
Parameters:
[1] 0.4963568 0.5201192 0.3734082 0.5862357
```

```
Iteration:  2 ,  3  function calls, F=  88.08288
Parameters:
[1]  0.6387613  0.5379012 -0.5332946  0.9828399
Iteration:  3 ,  5  function calls, F=  86.83833
...
Iteration:  14 ,  20  function calls, F= 86.08953
Parameters:
[1]  0.6336458  0.5637054 -0.1829599  0.9061303
> msobj$param
        b1         b2         b3         b4
 0.6336399 0.5637011 -0.1829409 0.9061293

> glm(cbind(rsp,1-rsp) ~ v1 + v2 + v3 + v4 + Int - 1,
         family=binomial(link=probit), data=data.frame(
         matrix(cbind(respLgst,rep(1,200),matcov), ncol=6,
         dimnames=list(NULL,c("rsp","v1","v2", "v3","v4","Int")))),
         start=c(matcov %*% rep(0,4)) - 2)
Error in glm.fitter: Missing value where logical needed:
if(df.residual > 0) fit$assign.residual <- (rank + 1):n
```

So all of the methods work well, except that *glm* specified with the wrong choice of link may not converge. With the correct choice of link, we have better luck:

```
> tmpglm <- glm(cbind(rsp,1-rsp) ~ v1 + v2 + v3 + v4 + Int - 1,
         family=binomial, data=data.frame(
         matrix(cbind(respLgst,matcov,rep(1,200)), ncol=6,
         dimnames=list(NULL,c("rsp","v1","v2", "v3","v4","Int")))),
         start=c(matcov %*% rep(0,4)) - 2)
> tmpglm$coef
        v1         v2         v3        v4        Int
 0.8695273 0.7992177 -0.449265 1.347821 -2.899334

> c(NRroot(rep(0,5), function(bb) t(Gradmat(bb, function(uu)
    -binregLik(uu[1:4],uu[5],matcov,respLgst,dist=plogis))) )$final)
[1]  0.8695275  0.7992178 -0.4492651  1.3478211 -2.8993347
```

Concerning the last model-fit, we digress momentarily: note that *glm* does not seem to give a way to fix the intercept and fit the logistic regression model using restricted Maximum Likelihood. In fact, there is a way (custom-modifying the link definition), but it is unreasonably difficult, so one would probaby use *ms* or some other estimation function instead. In the last-fitted model, we can examine the incremental deviances due to successively added model terms as follows:

```
> anova(tmpglm)
Analysis of Deviance Table,  Binomial model
Response: cbind(rsp, 1 - rsp)
Terms added sequentially (first to last)

     Df Deviance Resid. Df Resid. Dev
NULL                   200   277.2589
  v1  1 24.30848       199   252.9504
  v2  1  5.96717       198   246.9832
  v3  1 27.07522       197   219.9080
  v4  1  0.06075       196   219.8473
 Int  1 48.84046       195   171.0068
```

This says in particular that the intercept coefficient (entered last) is highly significant: p-value $= 1 - pchisq(48.84, 1) = 3e - 12$. The correct intercept value, we know, is -2, while the estimated value is -2.899, and the other coefficients are also different from the true ones ! However:

```
> -binregLik(tmpglm$coef[1:4],tmpglm$coef[5],matcov,respLgst)
[1] 85.5034                     ###  -logLik  for 5-par model
> -binregLik(c(NRroot(rep(0,4), function(bb) t(Gradmat(bb, function(bb)
    -binregLik(bb,-2,matcov,respLgst,dist=plogis))) )$final),-2,
    matcov,respLgst)
[1] 87.32924                    ### -logLik for 4-par model
```

So the likelihood ratio statistic (χ^2_1) for the difference of the Intercept coefficient from -2 with these data is $2 \cdot (87.329 - 85.503) = 3.472$ which gives p-value 0.062. Thus the difference between the fitted Interecept and -2, which looked striking, is nevertheless not significant.

6.2 *Statistical & Likelihood-based theory*

The optimization of likelihoods (and many other functions like *distance* or *contrast* functions between observations and theoretical expectations based on parametric models) are extremely special from the point of view of numerical optimization. The main point is that there is underlying theory to say that *if the underlying statistical model fits* then the locally quadric surface near a likelihood maximum has curvatures for which we have Fisher-information-related theoretical expressions which can be estimated ! This gives some sort of check that the correct local optimum has been reached.

Your Stat 700-701 books have material on MLE closely related to this topic. An additional reference at about the same level showing lots of examples involving local theory for MLE's is the book *Theoretical Statistics* of Cox & Hinkley. (I believe this book also has accessible discussion of misspecified models.) An important related paper is:

Efron, B. & Hinkley, D. (1978) Assessing the accuracy of the MLE: observed vs. expected Fisher information. Biometrika $\underline{65}$, $457 - 87$.

The message of the paper is primarily that it is better to use observed Fisher information of making confidence intervals from MLE's than is the theoretical Fisher Information with substituted parameter-estimators. But in our context, we should want to calculate and compare **both** in order to assess model-validity and correctness of convergence.

On the other hand, hypothesized models often turn out not to fit well, and this has consequences for the estimation of parameters via numerical maximization. We discussed above the checking of two kinds of 'expectedinformation' against the theoretical information matrix, with the numerically calculated MLE $\hat{\vartheta}$ substituted. It was mentioned that this is a little optimistic in the usual case where you have no real reason to know that the family of parametric models being fitted to the data is properly specified. In case the data are analyzed by optimizing loglikelihood $l(\underline{X}, \vartheta)$ with respect to a specific (but possibly wrong) model, it can still be shown under general conditions that there is an asymptotic value ϑ_* to which the MLE $\hat{\vartheta}$

converges, with

$$\hat{\vartheta} - \vartheta_* \;\approx\; -\Big(\nabla_\vartheta^{\otimes 2} l(\underline{X}, \vartheta_*)\Big)^{-1} \nabla_\vartheta l(\underline{X}, \vartheta_*)$$

where, for any vector v, the notation $v^{\otimes 2}$ denotes $v\, v^t$. Therefore, in the context of *iid* data with density f, we would want to compute confidence intervals for $\hat{\vartheta}$ not directly from any single observed or theoretical information but by treating the asymptotic variance-covariance of $\hat{\vartheta}$ as

$$\Big(\nabla_\vartheta^{\otimes 2} l(\underline{X}, \vartheta_*)\Big)^{-1} \; \sum_{i=1}^{n} (\nabla \log f(X_i, \vartheta_*))^{\otimes 2} \Big(\nabla_\vartheta^{\otimes 2} l(\underline{X}, \vartheta_*)\Big)^{-1}$$

In addition, an indication of lack of fit of a model with ML estimated parameter $\hat{\vartheta}$ (on which are based the *misspecification tests* used by econometricians) is a large discrepancy between any of

$$I(\hat{\vartheta}) \;=\; -\int \Big(\nabla_\vartheta^{\otimes 2} \log f(x, \vartheta)\Big)\; f(x, \vartheta)\, dx\, |_{\vartheta = \hat{\vartheta}}$$

$$\text{or} \qquad -\frac{1}{n} \sum_{i=1}^{n} \nabla_\vartheta^{\otimes 2} \log f(X_i, \hat{\vartheta}) \qquad \text{or} \qquad \frac{1}{n} \sum_{i=1}^{n} \Big(\nabla_\vartheta \log f(X_i, \hat{\vartheta})\Big)^{\otimes 2}$$

All of these, especially the first two, can be compared to check for correct maximization in any simulation from a model $f(x, \vartheta)$. However, in real-data settings, these matrices may be different *either* because the model is wrong *or* because convergence to the proper MLE has not taken place !

References for this topic include a famous 1967 Fifth Berkeley Symposium paper by Peter Huber and (a more recent paper which cites it) :

H. White (1982) Maximum likelihood estimation of misspecified models. *Econometrica* **50**, 1-25.

6.3 More on Numerical Maximization

6.3.1 *Methods with Constraints on Parameters*

- Re-parameterizations. For example, if a parameter λ is constrained to be positive, then it could be reparameterized as e^{ϑ} for an arbitrary real ϑ. Similarly, a probability parameter π constrained to be between 0, 1 could be re-defined as $\log(\frac{\pi}{1-\pi})$. The numerical maximization is then performed with the *unconstrained* parameter.
- Penalty functions to enforce box-constraints *(cf.* **nlminb***)*
- Projections to enforce functional constraints

For the latter two approaches, see Luenberger cited previously, or a numerical analysis text.

Example: 'Additive risk' model

Two-group data, with group-indicators z_i , and with observations which are Expon(λ) if $z_i = 0$ and Expon($\lambda + \alpha$) if $z_i = 1$, where both λ, $\alpha > 0$.

6.3.2 *Optimization Methods Using Randomness*

- Random-restart methods to check uniqueness of local maxima or global relative values
- Random perturbation methods, e.g. "Simulated Annealing"

References:

Kirkpatrick, S., Gelatt, C. & Vecchi, M. (1983) Optimization by simulated annealing. *Science* **220**, 671-80.

Geman, S. & Deman, D. (1984) Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.* **6**, 721-41.

6.4 Methods of Dealing with Missing Data

- Random multiple imputation
- EM algorithm: examples from contingency tables & mixture-data

References:

(1). Little, R. & Rubin, D. (1986) **Statistical Analysis of Missing Data**. Wiley.

(2). Dempster, A., Laird, N. & Rubin, D. (1978) Maximum likelihood from incomplete data via the EM algorithm. *Jour. Roy. Statist. Soc* **B 40**, 1-22.

(3). Wu, C.-F. (1983) *Ann. Stat.* **11**, 95-103.