**Optimization**

Objective

Determine what values for “inputs” of a function lead to the maximization or minimization of the function. Often, these inputs can be found through solving an equation (derived from the original function) set equal to 0.

Applied mathematics is the main research area for optimization. Because so many disciplines rely on optimization of a function, these other disciplines, including statistics, can advance the area as well. Primarily, most statisticians are “users” of optimization research, and I will approach these lecture notes in this way.

We will focus on numerical optimization.

Example: Maximum likelihood estimation for π

Suppose Y1, …, Yn are iid random variables with a Bernoulli(π) distribution. What is the maximum likelihood estimate (MLE) of π?

The likelihood function is



where  and . The function to be maximized is L(π|**y**) and the input is π.

In STAT 883, you derive the closed form expression for the MLE to be  = w/n. Suppose a closed form expression is not available. How would you numerically obtain the MLE?

Simple search algorithms

Sometimes, optimization problems are made more difficult than they need to be. For example, if one is looking for the maximum of a function, one could simply evaluate the function at a grid of inputs to find where the maximum occurs! This can even be especially helpful to validate a more complicated optimization method.

Example: Maximum likelihood estimation for Bernoulli (MLE\_Bernoulli.R)

Suppose w = 4 and n = 10. Define  Note that



> w <- 4

> n <- 10

> L <- function(pi, w, n) {

pi^w \* (1 - pi)^(n - w)

}

> logL <- function(pi, w, n) {

w \* log(pi) + (n - w) \* log(1 - pi)

}

> curve(expr = L(pi = x, w = w, n = n), xlim = c(0,1), xlab

= expression(pi), ylab = "Likelihood", col = "red")

> abline(v = w/n, lty = "dotted")

> curve(expr = logL(pi = x, w = w, n = n), xlim = c(0,1),

xlab = expression(pi), ylab = "log(L)", col = "red")

> abline(v = w/n, lty = "dotted")

> pi<-seq(from = 0.3, to = 0.5, by = 0.01)

> data.frame(pi = pi, L = L(pi = pi, w = w, n = n), logL =

logL(pi, w = w, n = n))

pi L logL

1 0.30 0.0009529569 -6.955941

2 0.31 0.0009966469 -6.911114

3 0.32 0.0010367007 -6.871712

4 0.33 0.0010727650 -6.837516

5 0.34 0.0011045345 -6.808331

6 0.35 0.0011317547 -6.783986

7 0.36 0.0011542233 -6.764328

8 0.37 0.0011717911 -6.749222

9 0.38 0.0011843622 -6.738551

10 0.39 0.0011918935 -6.732212

11 0.40 0.0011943936 -6.730117

12 0.41 0.0011919211 -6.732189

13 0.42 0.0011845820 -6.738365

14 0.43 0.0011725273 -6.748594

15 0.44 0.0011559495 -6.762833

16 0.45 0.0011350793 -6.781053

17 0.46 0.0011101812 -6.803232

18 0.47 0.0010815501 -6.829360

19 0.48 0.0010495062 -6.859436

20 0.49 0.0010143910 -6.893467

21 0.50 0.0009765625 -6.931472



In most other situations, you will need to pass into a function each observation in a data set rather than a single numerical value like w here. Below is how the functions could be re-written.

> y<-c(1, 0, 0, 1, 0, 1, 0, 0, 0, 1) #Possible y vector

> sum(y)

[1] 4

> L2 <- function(pi, y) {

#prod(dbinom(x = y, size = 1, prob = pi)) #alternative

prod(pi^y \* (1 - pi)^(1 - y))

}

> logL2 <- function(pi, y) {

#sum(dbinom(x = y, size = 1, prob = pi, log = TRUE))

#alternative

sum(y \* log(pi) + (1 - y) \* log(1 - pi))

}

> L2(pi = sum(y)/n, y = y)

[1] 0.001194394

> logL2(pi = sum(y)/n, y = y)

[1] -6.730117

> L(pi = w/n, w = w, n = n)

[1] 0.001194394

> logL(pi = w/n, w = w, n = n)

[1] -6.730117

The key is to take advantage of vectorized calculations rather than loops with for(). The prod() and sum() functions can then be used at the end.

We can also perform the function maximization by first finding the partial derivative of (π|w) with respect to π:



set this equal to 0, and solve for π. Thus, we want to find the “root” of this equation.

> partial.logL <- function(pi, w, n) {

w/pi - (n - w)/(1 - pi)

}

> curve(expr = partial.logL(pi = x, w = w, n = n), xlim =

c(0,1), xlab = expression(pi), ylab = "Partial

derivative of log(L)", col = "red")

> abline(v = w/n, lty = "dotted")

> abline(h = 0, lty = "dotdash")



> data.frame(pi = pi, L = partial.logL(pi = pi, w = w, n =

n))

pi L

1 0.30 4.7619048

2 0.31 4.2075736

3 0.32 3.6764706

4 0.33 3.1659882

5 0.34 2.6737968

6 0.35 2.1978022

7 0.36 1.7361111

8 0.37 1.2870013

9 0.38 0.8488964

10 0.39 0.4203447

11 0.40 0.0000000

12 0.41 -0.4133940

13 0.42 -0.8210181

14 0.43 -1.2239902

15 0.44 -1.6233766

16 0.45 -2.0202020

17 0.46 -2.4154589

18 0.47 -2.8101164

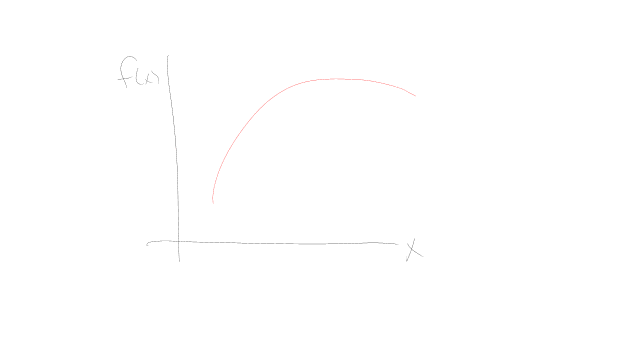
19 0.48 -3.2051282

20 0.49 -3.6014406

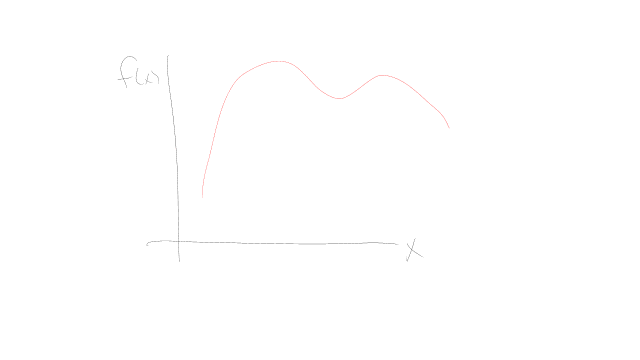
21 0.50 -4.0000000

Convex functions

Many optimization algorithms rely on having a convex function in order to guarantee finding the maximum or minimum. The function f(x) is convex if a line segment connecting any two points x1 and x2 is above (or below) a plot of the function. For example,



is convex, but



is not. The problem with having a non-convex function is that a local maximum, rather than the global maximum may be found by an optimization method.

The definition of a convexity extends as would be expected to a function of more than one input.

Newton-Raphson method

Suppose we would like to find the x such that f(x) = 0. Thus, we want to find the x-intercept for the function. This is the case when we set a partial derivative of a log-likelihood function equal to 0 and solve for a parameter when performing maximum likelihood estimation.

We can approximate f(x) by using a simple first-order Taylor series expansion



where x0 could be an initial guess for the root and  Notice this is a linear approximation to f(x), so it produces a simply linear line. Of course, if f(x) was a simple linear function, the approximation would be perfect.

A “better” guess for the root than x0 is obtained by choosing an x, say x1, such that



In other words, this is the x-intercept – where the linear line hits the x-axis on the plot. This value of x1 is obtained by solving the above expression for x1:



Using this x1, we can obtain a “better” guess again by choosing an x2 such that



This leads to an x2 value of



The process can continue using



in general. Once  for some very small positive constant ε, the sequence of roots is said to have converged to the root of f(x).

Example: Simple Newton-Raphson demo (Newton\_demo.R)

Suppose the function of interest is

f(x) = x3 – 5

for x = 1 to 4.5. A simple plot will show this function is convex within this range. The first derivative of f(x) is



Below is my code and output showing the first step (iteration) of the Newton-Raphson method:

> fx <- function(x, a = 5) {

x^3 - a

}

> curve(expr = fx(x), xlim = c(1, 4.5), col = "red", lwd =

2, ylim = c(-10, 100), xlab = "x", ylab = "f(x)")

> abline(h = 0, lty = "dotted")

> fx.dot <- function(x) {

3\*x^2

}

> fx.app <- function(x, x.guess) {

fx(x.guess) - fx.dot(x.guess)\*x.guess +

fx.dot(x.guess)\*x

}

> x.val <- function(x.guess) {

x.guess - fx(x.guess)/fx.dot(x.guess)

}

> #colors

> color.choice <- c(paste(rep("darkgoldenrod",4), 1:4, sep

= ""), "brown1", "brown4")

> #######################################################

> x.guess <- 4 #My initial choice

> abline(a = fx(x.guess) - fx.dot(x.guess) \* x.guess, b =

fx.dot(x.guess), lty = "dashed", col = color.choice[1])

> x.val(x.guess) #New x from linear approximation

[1] 2.770833

> fx.app(x = x.val(x.guess), x.guess = x.guess) #Verify

fx.app(new x) = 0

[1] 0

> x.guess <- x.val(x.guess) #Set old x to new x



Steps 2 through 6 produce the following (see code in program):



The most often use of the Newton-Raphson method for statisticians is maximum likelihood estimation. For this situation, the “f(x)” is the first partial derivative of the log-likelihood function. For a one parameter case θ, this leads to



for a scalar parameter θ and



This leads to using



When convergence is obtained, the last θ calculated is the MLE, denoted by .

Example: Maximum likelihood estimation for Bernoulli (MLE\_Bernoulli.R)

Suppose w = 4 and n = 10. Then



To begin using the Newton-Raphson method, we need to choose a starting value π0. Suppose this value is π0 = 0.3. Below is the Newton-Raphson method code in R:

> w <- 4

> n <- 10

> #Initialize some values

> epsilon<-0.0001 #Convergence criteria

> pi.hat<-0.3 #Start value

> save.pi.hat<-pi.hat #Save the results for each iteration

here and put pi.hat as first value

> change<-1 #Initialize change for first time

through loop

> #Loop to find the MLE (uses Newton-Raphson algorithm)

> while (abs(change) > epsilon) {

num<-(w-n\*pi.hat) / (pi.hat\*(1-pi.hat))

den<- -w/pi.hat^2 - (n-w)/(1-pi.hat)^2

pi.hat.new<-pi.hat - num/den

change<-pi.hat.new-pi.hat #-num/den

pi.hat<-pi.hat.new #Same for next time through loop

save.pi.hat<-c(save.pi.hat, pi.hat) #Keeps iteration

history

}

> #Print iteration history

> data.frame(iteration = 1:length(save.pi.hat),

save.pi.hat)

iteration save.pi.hat

1 1 0.3000000

2 2 0.3840000

3 3 0.3997528

4 4 0.3999999

5 5 0.4000000

Iteration #5 is where convergence is obtained. Below are some additional plots that demonstrate the process:







When **θ** = (θ1, …, θp)′, we have



where  is a p×1 vector of partial derivatives and  is a p×p matrix of second partial derivatives evaluated at **θ**0. The structure of  is



Matrices of second partial derivatives like this are called “Hessian” matrices. The iterative method uses



where the MLE obtained at convergence is denoted by 

Fisher scoring is a quasi-Newton Raphson method that uses the same iterative procedure, but with the Hessian matrix replaced with its expected value. Thus, use  instead of . The expected value can be computationally easier to compute (although more iterations may be necessary).

Comments:

* Obviously, we are assuming that  is twice differentiable with a non-zero derivative.
* When  does not depend on **y**,  = . For example, this occurs with generalized linear models when the canonical link function is used.
*  is Fisher’s information matrix! Note that for a maximum likelihood estimator **θ**, we have



*  (see top of p. 474 of Casella and Berger 2002). Thus,  is often taken to approximate , and it is often referred to as the “observed” Fisher information. This is very convenient because R can numerically differentiate  so that one does not need to code in R the Hessian matrix!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Of course, numerical derivatives are not necessarily the same as the actual derivatives.
* An initial choice for the MLEs is needed to begin the Newton-Raphson method. Possible choices include:
  + Method of moment (MOM) estimates
  + MLEs from simpler models; e.g., use normal linear regression estimates to approximate more complicated regression model parameter estimates
* Problems:
  + There may be more than one “maximum” value of a likelihood function
  + Minimum rather than a maximum?

R functions for optimization

A task view is dedicated to these functions at <http://cran.r-project.org/web/views/Optimization.html>.

We are going to focus on using the optim() function within the stats package. This function implements optimization methods that are similar to the Newton-Raphson method along with a few other methods as well. The nice aspects of using this function include:

* You only need to provide the mathematical function to be optimized! While you can also provide derivatives, these are not necessary (although it can help).
* A numerical estimate of the Hessian matrix can be provided by the function.

Please note that this function actually performs minimization rather than maximization. This is not limiting because maximizing f(x) is equivalent to minimizing –f(x).

Example: Maximum likelihood estimation for parameters from a gamma distribution (MLE\_gamma.R)

Using the Casella and Berger (2002) parameterization, below are the likelihood and log-likelihood functions:





Initial estimates for the parameters can be found by using the MOMs:

For the gamma distribution, we first set the mean and variance equal to their corresponding moment estimates:

 and 

where . Next, we solve for α and β to obtain the MOMs:

 and 

Below is my simulated data from a gamma(3.5, 2):

> set.seed(4869)

> n <- 30

> alpha <- 7/2

> beta <- 2 #Chisq(7)

> y <- rgamma(n = n, shape = alpha, scale = beta)

> round(y,2)

[1] 3.61 10.08 1.90 3.75 10.43 5.52 9.86 1.84 12.53

[10] 6.09 12.40 3.31 16.48 6.34 6.65 10.11 9.18 7.98

[19] 1.84 2.14 3.65 10.34 10.88 14.31 5.82 10.70 1.58

[28] 13.48 5.37 11.31

Plots (code in program):



Below is how the optim() function is can be used.

> logL1 <- function(theta, y) {

n <- length(y)

alpha <- theta[1]

beta <- theta[2]

sum(dgamma(x = y, shape = alpha, scale = beta, log =

TRUE))

}

> alpha.mom <- mean(y)^2/((n-1)\*var(y)/n)

> beta.mom <- ((n-1)\*var(y)/n) /mean(y)

> data.frame(alpha.mom, beta.mom)

alpha.mom beta.mom

1 3.359576 2.355641

> gam.mle1 <- optim(par = c(alpha.mom, beta.mom), fn =

logL1, y = y, method = "BFGS", control = list(fnscale =

-1), hessian = TRUE)

> gam.mle1

$par

[1] 2.661508 2.874372

$value

[1] -84.80431

$counts

function gradient

24 8

$convergence

[1] 0

$message

NULL

$hessian

[,1] [,2]

[1,] -13.64773 -10.437065

[2,] -10.43706 -9.664134

> cov.mat <- -solve(gam.mle1$hessian)

> cov.mat

[,1] [,2]

[1,] 0.4208905 -0.4545530

[2,] -0.4545530 0.5943832

> data.frame(name = c("alpha", "beta"), estimate =

gam.mle1$par, SE = sqrt(diag(cov.mat)))

name estimate SE

1 alpha 2.661508 0.6487607

2 beta 2.874372 0.7709625

The maximum likelihood estimates are  and . The maximum of the likelihood function is   
-84.80.

Comments:

* Many R functions for densities have a log argument. One reason is because of maximum likelihood estimation. Notice how it was rather simple to specify the likelihood function in logL1()!
* The method = "BFGS" argument corresponds to the type of optimization used. This is essentially the Newton-Raphson method where an approximation is made to the Hessian matrix. This approximation can then make the computations easier.The BFGS stands for “Broyden-Fletcher-GoldFarb-Shanno”. Further details are given on p. 165 of Bloomfield (2014) and p. 42 of Givens and Hoeting (2013)
* The control argument allows various options to be stated. In particular, fnscale = -1 corresponds to scaling the function to be minimized by -1. Thus, we are minimizing the negative of the returned value by logL1().
* The 0 value in the convergence component of the returned list means that convergence occurred. A value of 1 means non-convergence. There are other values that can occur, and these are described in the help for optim().
* The large sample distribution for the MLEs is estimated to be



How could we find a confidence interval for α or β using this information? How could we use MC simulation to examine if this distribution really was appropriate for n = 30?

For problems like this, it is helpful to view the log-likelihood function to verify that the maximum has truly been found. Even for more complex problems, plots for simpler cases should be tried as a way to check the calculations. Below are plots of the log-likelihood function.

> alpha<-seq(0.25,10,0.1)

> beta<-seq(0.25,10,0.1)

> pars<-as.matrix(expand.grid(alpha, beta))

> #Evaluate logL values

> loglik<-matrix(data = NA, nrow = nrow(pars), ncol = 1)

> for(i in 1:nrow(pars)){

theta<-pars[i,]

loglik[i,1]<-c(logL1(theta = theta, y = y))

}

> #Put log(L) into matrix - need to be careful about

ordering

> loglik.mat<-matrix(data = loglik, nrow = length(alpha),

ncol = length(beta), byrow = FALSE)

> head(loglik)

[,1]

[1,] -987.5555

[2,] -967.2877

[3,] -949.8982

[4,] -934.3219

[5,] -920.0228

[6,] -906.6901

> loglik.mat[1:5, 1:5]

[,1] [,2] [,3] [,4] [,5]

[1,] -987.5555 -727.7880 -583.9556 -492.7314 -429.7872

[2,] -967.2877 -708.5295 -565.4510 -474.8289 -412.3859

[3,] -949.8982 -692.1495 -549.8249 -459.8048 -397.8629

[4,] -934.3219 -677.5825 -536.0120 -446.5938 -385.1531

[5,] -920.0228 -664.2930 -523.4763 -434.6602 -373.7206

> logL1(theta = c(0.25, 0.25), y = y)

[1] -987.5555

> logL1(theta = c(0.25, 0.35), y = y)

[1] -727.788

> logL1(theta = c(0.35, 0.25), y = y)

[1] -967.2877

> par(pty = "s")

> contour(x = alpha, y = beta, z = loglik.mat, levels = c(-

85, -86, -88, -90, -100, -200, -300, -400), xlab =

expression(alpha), ylab = expression(beta), main =

"Contour plot of log(L)")

> abline(h = gam.mle1$par[2], col = "blue")

> abline(v = gam.mle1$par[1], col = "blue")

> max(loglik)

[1] -84.81105

> #From help for contour: Notice that contour interprets

the z matrix as a table of f(x[i], y[j]) values,

> # so that the x axis corresponds to row number and the y

axis to column number



> library(package = rgl) #Package that does 3D interactive

plots

> open3d() #Open plot window

wgl

3

> #3D plot with gridlines

> persp3d(x = alpha, y = beta, z = loglik.mat, xlab =

"alpha", ylab = "beta", zlab = "log(L)", ticktype =

"detailed", col = "red")

> spheres3d(x = gam.mle1$par[1], y = gam.mle1$par[2], z =

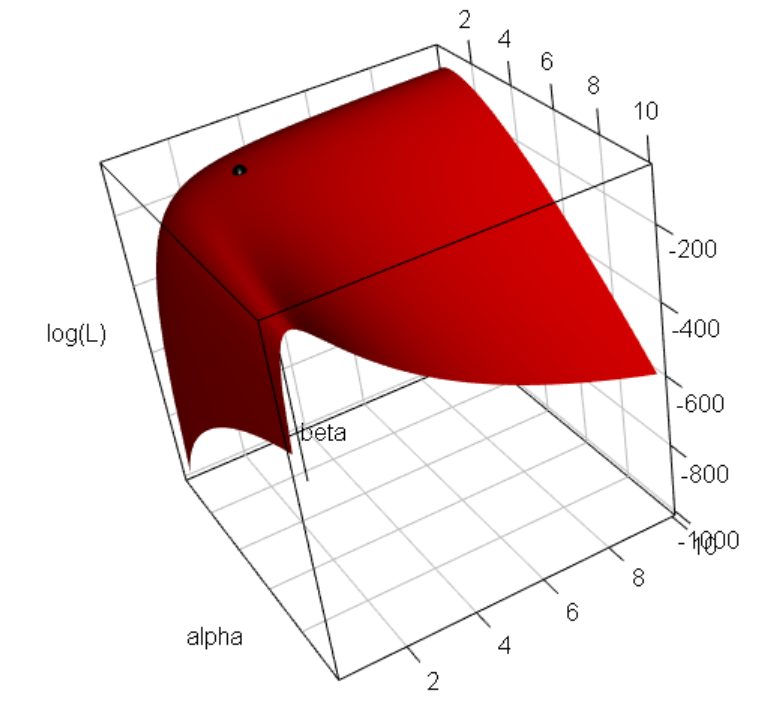
max(loglik), radius = 10)

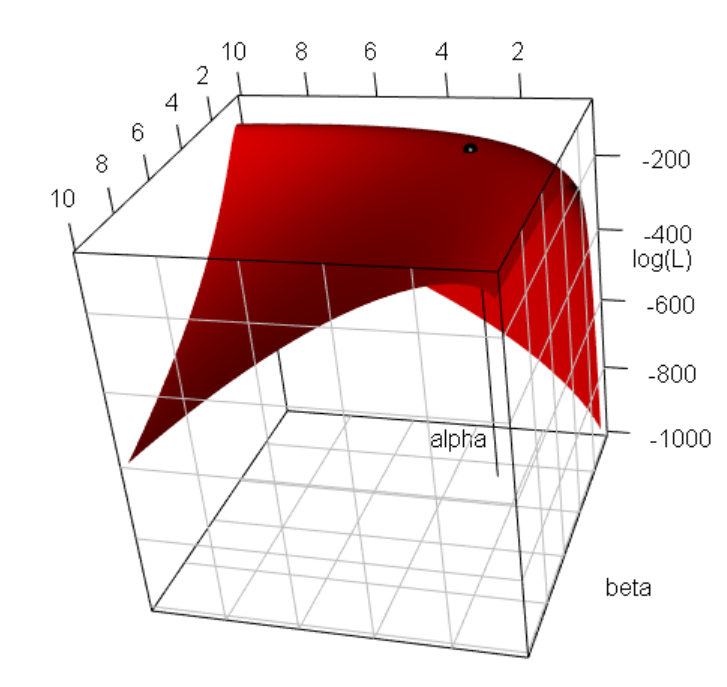
> lines3d(x = c(gam.mle1$par[1], gam.mle1$par[1]), y =

c(gam.mle1$par[2], gam.mle1$par[2]), z = c(min(loglik),

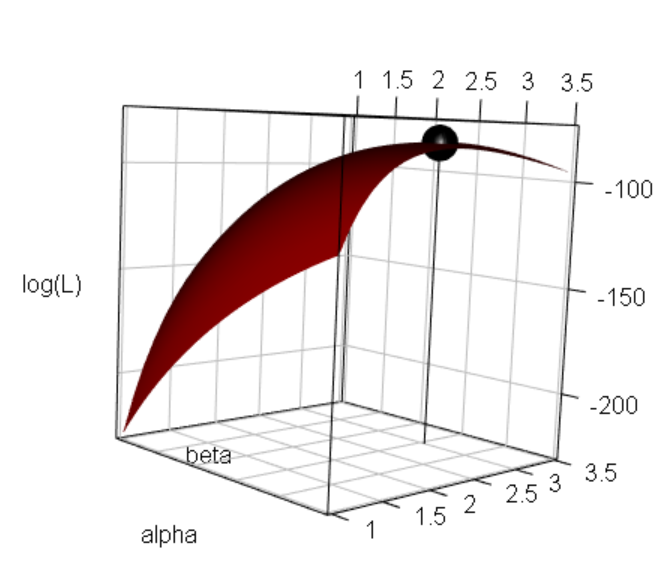
max(loglik)))

> grid3d(c("x", "y+", "z"))





Using some additional code in the program, I looked at the 3D plot zoomed in:



The parameters from a gamma distribution are defined to be positive. Notice that my implementation of the optim() function did not include this restriction. There are a few ways to handle this situation (other than “hope” no problems occur ☺)

1. Use method = "L-BFGS-B" and add a lower and upper argument to give the constraints on the parameters:

> gam.mle1.constr <- optim(par = c(alpha.mom, beta.mom),

fn = logL1, y = y, method = "L-BFGS-B", control =

list(fnscale = -1), hessian = TRUE, lower = c(0, 0),

upper = c(Inf, Inf))

> gam.mle1.constr

$par

[1] 2.661522 2.874372

$value

[1] -84.80431

$counts

function gradient

10 10

$convergence

[1] 0

$message

[1] "CONVERGENCE: REL\_REDUCTION\_OF\_F <= FACTR\*EPSMCH"

$hessian

[,1] [,2]

[1,] -13.64764 -10.437064

[2,] -10.43706 -9.664078

1. Find the MLEs for the log of the parameters first and then use the exponential function to transform back to the original numerical scale. Because the exponential function is always positive, the estimates will be positive. Plus, they will be MLEs due to the invariance property of MLEs. The multivariate δ-method then leads to the estimated covariance matrix.

The δ-method (see p. 240 of Casella and Berger 2002) says that if



then



The multivariate δ-method states that if



then



For the gamma distribution setting here, I will find the MLEs for  and . The MLEs for α and β are  and  respectively. Below is my code and output:

> logL2 <- function(theta, y) {

n <- length(y)

alpha <- exp(theta[1])

beta <- exp(theta[2])

sum(dgamma(x = y, shape = alpha, scale = beta, log =

TRUE))

}

> gam.mle2 <- optim(par = log(c(alpha.mom, beta.mom)), fn =

logL2, y = y, method = "BFGS", control = list(fnscale =

-1), hessian = TRUE)

> gam.mle2

$par

[1] 0.9788797 1.0558605

$value

[1] -84.80431

$counts

function gradient

15 6

$convergence

[1] 0

$message

NULL

$hessian

[,1] [,2]

[1,] -96.6753 -79.8442

[2,] -79.8442 -79.8431

> exp(gam.mle2$par)

[1] 2.661473 2.874447

> #Estimated covariance matrix

> cov.mat.log <- -solve(gam.mle2$hessian)

> cov.mat.log

[,1] [,2]

[1,] 0.05941768 -0.05941850

[2,] -0.05941850 0.07194388

> gdot <- matrix(data = c(exp(gam.mle2$par[1]), 0,

0, exp(gam.mle2$par[2])), nrow =

2, ncol = 2)

> cov.mat2 <- gdot %\*% cov.mat.log %\*% gdot

> cov.mat2

[,1] [,2]

[1,] 0.4208814 -0.4545672

[2,] -0.4545672 0.5944326

> data.frame(name = c("alpha", "beta"), estimate =

exp(gam.mle2$par), SE = sqrt(diag(cov.mat2)))

name estimate SE

1 alpha 2.661473 0.6487538

2 beta 2.874447 0.7709946

Comments:

* + - Always try to use functions available in R for distributions rather than programming in the distribution equations yourself. My program gives an example of the problems that can occur when dgamma() is not used.
    - constrOptim() could also be used to set constraints on the parameters.
    - The above use of optim() could be extended to a generalized linear model setting where Y ~ gamma and E(Y) is a function of explanatory variables. Note that a reparameterization of the gamma distribution leads to

 for y, κ, μ > 0.

With this formulation of the distribution, E(Y) = μ and Var(Y) = μ2/κ. This is different from the usual definition of a gamma distribution in Casella and Berger (2002), where E(Y) = αβ and Var(Y) = αβ2. Thus, the relationship between the two definitions results in β = μ/κ and α = κ.

Other useful R functions

* nlm() and nlminb() – Function optimization using a method similar to Newton-Raphson.
* uniroot() – Useful to find a single root of an equation.
* optimize() – One-dimensional unconstrained function optimization
* maxlik package – Implements more methods similar to Newton-Raphson
* optimx package – An attempt to unify the R functions for optimization