Maximum Likelihood Programming in R

Marco R. Steenbergen
Department of Political Science
University of North Carolina, Chapel Hill

January 2006

Contents

1 Introduction

The programming language R is rapidly gaining ground among political methodologists. A major reason is that R is a flexible and versatile language, which makes it easy to program new routines. In addition, R algorithms are generally very precise.

R is well-suited for programming your own maximum likelihood routines. Indeed, there are several procedures for optimizing likelihood functions. Here I shall focus on the `optim` command, which implements the BFGS and L-BFGS-B algorithms, among others.[1] Optimization through `optim` is relatively straightforward, since it is usually not necessary to provide analytic first and second derivatives. The command is also flexible, as likelihood functions can be declared in general terms instead of being defined in terms of a specific data set.

2 Syntactic Structure

Estimating likelihood functions entails a two-step process. First, one declares the log-likelihood function, which is done in general terms. Then one optimizes the log-likelihood function, which is done in terms of a particular data set. The log-likelihood function and optimization command may be typed interactively into the R command window or they may be contained in a text file. I would recommend saving log-likelihood functions into a text file, especially if you plan on using them frequently.

2.1 Declaring the Log-Likelihood Function

The log-likelihood function is declared as an R `function`. In R, functions take at least two arguments. First, they require a vector of parameters. Second, they require at least one data object. Note that other arguments can be added to this if they are necessary. The data object is a generic placeholder for data. In the `optim` command, specific data are substituted for this placeholder.

After the arguments are declared, the actual log-likelihood is expressed and demarcated by {}. Thus, we have the following syntax:

```
name<-function(pars,object){
declarations
logl<-loglikelihood function
return(-logl)
}
```

Here `name` is the name of the log-likelihood function, `pars` is the name of the parameter vector, and `object` is the name of the generic data object. The instructions placed between brackets define the log-likelihood function. At a minimum, there should be two elements here: (1) the declaration of the log-likelihood

[1] The `optim` command also includes Nelder-Mead, conjugate gradients, and simulated annealing algorithms. Other optimization routines include `optimize`, `nlm`, and `constrOptim`. These procedures are not discussed here.

function, which is named `logl`, and (2) the return of negative one times the log-likelihood.[2] In addition, it may be necessary to make other `declarations`. These may include partitioning a parameter vector or declaring temporary variables that figure in the log-likelihood function. The application of this syntax will be clarified using several examples.

Example 1: Consider the Poisson log-likelihood function, which is given by

$$l = \sum_i y_i \ln(\mu) - n\mu - \sum_i \ln(y_i!)$$

Since the last term does not include the parameter, μ, it can be safely ignored. Thus, the kernel of the log-likelihood function is

$$l = \sum_i y_i \ln(\mu) - n\mu$$

We can program this function using the following syntax:

```
poisson.lik<-function(mu,y){
n<-nrow(y)
logl<-sum(y)*log(mu)-n*mu
return(-logl)
}
```

Here `poisson.lik` is the name of the log-likelihood function; this name will be used in the `optim` command. The "vector" of parameters is called `mu`; this is not really a vector since there is only one parameter that needs to be estimated. Further, `y` is the placeholder for the data. Since the log-likelihood function requires knowledge of the sample size, we obtain this using `n<-nrow(y)`. The expression for `logl` contains the kernel of the log-likelihood function. Finally, we ask R to return -1 times the log-likelihood function.

Example 2: Imagine that we have a sample that was drawn from a normal distribution with unknown mean, μ, and variance, σ^2. The objective is to estimate these parameters. The normal log-likelihood function is given by

$$l \quad = \quad -.5n\ln(2\pi) - .5n\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_i (y_i - \mu)^2$$

We can program this function in the following way:

```
normal.lik1<-function(theta,y){
mu<-theta[1]
sigma2<-theta[2]
n<-nrow(y)
```

[2]We ask for $-1 \times l$ because the `optim` command minimizes a function by default. Minimization of $-l$ is the same as maximization of l, which is what we want.

```
logl<- -.5*n*log(2*pi) -.5*n*log(sigma2) -
(1/(2*sigma2))*sum((y-mu)**2)
return(-logl)
}
```

Here `theta` is a vector containing the two parameters of interest. We declare the elements of this vector in the first two lines of the bracketed part of the program. Specifically, the first element (`theta[1]`) is equal to μ, while the second element (`theta[2]`) is equal to σ^2. The remainder of the program sets the sample size, specifies the log-likelihood function, and asks R to return the negative of this function.

Note that the normal log-likelihood function may also be written as

$$l \quad = \quad -n\ln(\sigma) + \sum_i \ln[\phi(z_i)]$$

where $z_i = (y_i - \mu)/\sigma$. This can be programmed using

```
normal.lik2<-function(theta,y){
mu<-theta[1]
sigma<-theta[2]
n<-nrow(y)
z<-(y-mu)/sigma
logl<- -n*log(sigma) - sum(log(dnorm(z)))
return(-logl)
}
```

where `dnorm` is R's standard normal density function. Here we estimate σ rather than σ^2, but it is easy to move back and forth between these parameterizations.

2.2 Optimizing the Log-Likelihood

Once the log-likelihood function has been declared, then the `optim` command can be invoked. The minimal specification of this command is

```
optim(starting values, log-likelihood, data)
```

Here `starting values` is a vector of starting values, `log-likelihood` is the name of the log-likelihood function that you seek to maximize, and `data` declares the data for the estimation. This specification causes R to use the Nelder-Mead algorithm. If you want to use the BFGS algorithm you should include the `method="BFGS"` option. For the L-BFGS-B algorithm you should declare `method="L-BFGS-B"`. The current specification does not produce standard errors. A procedure for obtaining standard errors will be discussed later in this report.[3]

[3]There are many other options for the `optim` command. For a detailed description see, for example, `http://jsekhon.fas.harvard.edu/stats/html/optim.html`.

Example 3: Imagine that we have a vector `data` that consists of draws from a Poisson distribution with unknown μ. We seek to estimate this parameter and have already declared the log-likelihood function as `poisson.lik`. Estimation using the BFGS algorithm now commences as follows

```
optim(1,poisson.lik,y=data,method="BFGS")
```

Here 1 is the starting value for the algorithm. Since the log-likelihood function refers to generic data objects as `y`, it is important that the vector `data` is equated with `y`.

Example 4: Given a vector of data, `y`, the parameters of the normal distribution can be estimated using

```
optim(c(0,1),normal.lik1,y=y,method="BFGS")
```

This is similar to Example 3 with the exception of the starting values. Since the normal distribution contains two parameters, two starting values need to be declared. Here we set the starting value for $\hat{\mu}$ to 0 and the starting value for $\hat{\sigma}^2$ to 1. These two values are "bundled" using the `c` or concatenation operator.

3 Output

The `optim` specifications discussed so far will produce several pieces of output. These come under various headings:

1. `$par`: This shows the MLEs of the parameters.
2. `$value`: This shows the value of the log-likelihood function at the MLEs. If you asked R to return -1 times the log-likelihood function, then this is the value reported here.
3. `$counts`: A vector that reports the number of calls to the log-likelihood function and the gradient.
4. `$convergence`: A value of 0 indicates normal convergence. If you see a 1 reported, this means that the iteration limit was exceeded. This limit is set to 10000 by default.
5. `$message`: This shows warnings of any problems that occurred during optimization. Ideally, one would like to see `NULL` here, since this indicates that there are no warnings.

4 Obtaining Standard Errors

The `optim` command allows one to compute standard errors based on the observed Fisher information matrix.[4] This requires that we obtain the Hessian

[4]Unlike Stata, standard errors, test statistics, and confidence intervals are not computed by default in the `optim` command.

matrix, which can be done by adding `hessian=T` or `hessian=TRUE` to the command. Since we will have to perform operations on the Hessian, it is also important that we store the results from the estimation into an object. The following linear regression example illustrates how to do this.

Example 5: Imagine that we are interested in estimating a simple linear regression for some simulated data. First, we create the data matrix for the predictors

```
X<-cbin(1,runif(100))
```

Here we draw 100 observations from a uniform distribution with limits 0 and 1. These data are bound together with the constant (1). Next, we postulate a set of values for the true parameters:

```
theta.true<-c(2,3,1)
```

Here, the first element is β_0, the second element is β_1, and the last element is σ^2. We can now create the dependent variable:

```
y<-X%*%theta.true[1:2] + rnorm(100)
```

where `rnorm(100)` generates the disturbance by drawing 100 values from the standard normal distribution. We now have the data on the dependent variable and predictor.

The next step is to declare the log-likelihood function. The following syntax shows one way to do this.

```
ols.lf<-function(theta,y,X){
n<-nrow(X)
k<-ncol(X)
beta<-theta[1:k]
sigma2<-theta[k+1]
e<-y-X%*%beta
logl<- -.5*n*log(2*pi)-.5*n*log(sigma2)-
((t(e)%*%e)/(2*sigma2))
return(-logl)
}
```

Here `theta` contains both the elements of $\boldsymbol{\beta}$ and σ^2. The program declares the first k elements of `theta` to be $\boldsymbol{\beta}$ and the $k+1$st element to be σ^2. The vector `e` contains the residuals and `t(e)%*%e` in the log-likelihood function causes R to compute the sum of squared residuals.

We can now start the optimization of the log-likelihood function and store the results in an object named `p` (any other name would have worked just as well):

```
p<-optim(c(1,1,1),ols.lf,method="BFGS",hessian=T,y=y,X=X)
```

where `c(1,1,1)` sets the starting values for $\hat{\beta}_0$, $\hat{\beta}_1$, and $\hat{\sigma}^2$ equal to 1. We can now invert the Hessian to obtain the observed Fisher information matrix.[5] This Hessian is stored as `p$hessian` and it can be inverted using

```
OI<-solve(p$hessian)
```

The square root of the diagonal elements are then the standard errors, corresponding to $\hat{\beta}_0$, $\hat{\beta}_1$, and $\hat{\sigma}^2$, respectively. These can be obtained by typing

```
se<-sqrt(diag(OI))
```

5 Test Statistics and Output Control

With the standard errors in hand, Wald test statistics and their associated p-values can be computed. The following syntax will accomplish this task for the regression model of Example 5.

Example 5 Cont'd: The Wald test statistic is given by the ratio of the estimates and their standard errors. The associated p-value can be computed by referring to a Student's t-distribution with degrees of freedom equal to the number of rows minus the number of columns in `X`.

```
t<-p$par/se
pval<-2*(1-pt(abs(t),nrow(X)-ncol(X)))
results<-cbind(p$par,se,t,pval)
results(colnames)<-c("b","se","t","p")
results(rownames)<-c("Const","X1","Sigma2")
print(results,digits=3)
```

The first line generates the test statistics, the second line computes the associated p-values, while the third line brings together the estimates, estimated standard errors, test statistics, and p-values. The fourth line creates a set of column headers for the output, while the fifth line creates a set of row headers. Finally, the last line causes R to print the results to the screen, with a precision of 3 digits.

[5] The observed Fisher information is equal to $(-\mathbf{H})^{-1}$. The reason that we do not have to multiply the Hessian by -1 is that all of the evaluation has been done in terms of -1 times the log-likelihood. This means that the Hessian that is produced by `optim` is already multiplied by -1.