Maximum Likelihood Estimation with *maxLik*

Ott Toomet

March 24, 2024

1 Introduction

This vignette is intended for users who are familiar with concepts of likelihood and with the related methods, such as information equality and BHHH approximation, and with `R` language. The vignette focuses on `maxLik` usage and does not explain the underlying mathematical concepts. Potential target group includes researchers, graduate students, and industry practitioners who want to apply their own custom maximum likelihood estimators. If you need a refresher, consult the accompanied vignette "Getting started with maximum likelihood and `maxLik`".

The next section introduces the basic usage, including the `maxLik` function, the main entry point for the package; gradients; different optimizers; and how to control the optimization behavior. These are topics that are hard to avoid when working with applied ML estimation. Section 3 contains a selection of more niche topics, including arguments to the log-likelihood function, other types of optimization, testing condition numbers, and constrained optimization.

2 Basic usage

2.1 The maxLik function

The main entry point to `maxLik` functionality is the function of the same name, `maxLik`. It is a wrapper around the underlying optimization algorithms that ensures that the returned object is of the right class so one can use the convenience methods, such as `summary` or `logLik`. It is important to keep in mind that `maxLik` *maximizes*, not minimizes functions.

The basic usage of the function is very simple: just pass the log-likelihood function (argument `logLik`) and the start value (argument `start`). Let us demonstrate the basic usage by estimating the normal distribution parameters. We create 100 standard normals, and estimate the best fit mean and standard deviation. Instead of explicitly coding the formula for log-likelihood, we rely on the `R` function `dnorm` instead (see Section 2.3 for a version that does not use `dnorm`):

```
> x <- rnorm(100)  # data.  true mu = 0, sigma = 1
> loglik <- function(theta) {
+    mu <- theta[1]
+    sigma <- theta[2]
+    sum(dnorm(x, mean=mu, sd=sigma, log=TRUE))
```

```
+ }
> m <- maxLik(loglik, start=c(mu=1, sigma=2))
>                             # give start value somewhat off
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 7 iterations
Return code 1: gradient close to zero (gradtol)
Log-Likelihood: -144.6843
2  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.01021    0.10282  -0.099   0.921
sigma  1.02830    0.07271  14.142  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

The algorithm converged in 7 iterations and one can check that the results are equal to the sample mean and variance.[1]

This example demonstrates a number of key features of `maxLik`:

- The first argument of the likelihood must be the parameter vector. In this example we define it as $\boldsymbol{\theta} = (\mu, \sigma)$, and the first lines of `loglik` are used to extract these values from the vector.
- The `loglik` function returns a single number, sum of individual log-likelihood contributions of individual x components. (It may also return the components individually, see BHHH method in Section 2.3 below.)
- Vector of start values must be of correct length. If its components are named, those names are also displayed in `summary` (and for `coef` and `stdEr`, see below).
- `summary` method displays a handy summary of the results, including the convergence message, the estimated values, and statistical significance.
- `maxLik` (and other auxiliary optimizers in the package) is a *maximizer*, not minimizer.

As we did not specify the optimizer, `maxLik` picked Newton-Raphson by default, and computed the necessary gradient and Hessian matrix numerically.

Besides summary, `maxLik` also contains a number of utility functions to simplify handling of estimated models:

- `coef` extracts the model coefficients:

```
> coef(m)
```

[1] Note that `R` function `var` returns the unbiased estimator by using denominator $n - 1$, the ML estimator is biased with denominator n.

```
         mu       sigma
-0.01020676  1.02829708
```

- `stdEr` returns the standard errors (by inverting Hessian):

```
> stdEr(m)

        mu      sigma
0.10282134 0.07271113
```

- Other functions include `logLik` to return the log-likelihood value, `returnCode` and `returnMessage` to return the convergence code and message respectively, and `AIC` to return Akaike's information criterion. See the respective documentation for more information.
- One can also query the number of observations with `nObs`, but this requires likelihood values to be supplied by observation (see the BHHH method in Section 2.3 below).

2.2 Supplying analytic gradient

The simple example above worked fast and well. In particular, the numeric gradient `maxLik` computed internally did not pose any problems. But users are strongly advised to supply analytic gradient, or even better, both the gradient and the Hessian matrix. More complex problems may be intractably slow, converge to a sub-optimal solution, or not converge at all if numeric gradients are noisy. Needless to say, unreliable Hessian also leads to unreliable inference. Here we show how to supply gradient to the `maxLik` function.

We demonstrate this with a linear regression example. Non-linear optimizers perform best in regions where level sets (contours) are roughly circular. In the following example we use data in a very different scale and create the log-likelihood function with extremely elongated elliptical contours. Now Newton-Raphson algorithm fails to converge when relying on numeric derivatives, but works well with analytic gradient.

We combine three vectors, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$, created at a very different scale, into the design matrix $\mathsf{X} = \begin{pmatrix}\boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3\end{pmatrix}$ and compute $\boldsymbol{y}$ as

$$\boldsymbol{y} = \mathsf{X}\begin{pmatrix}1\\1\\1\end{pmatrix} + \boldsymbol{\epsilon}. \tag{1}$$

We create $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$ as random normals with standard deviation of 1, 1000 and 10^7 respectively, and let $\boldsymbol{\epsilon}$ be standard normal disturbance term:

```
> ## create 3 variables with very different scale
> X <- cbind(rnorm(100), rnorm(100, sd=1e3), rnorm(100, sd=1e7))
> ## note: correct coefficients are 1, 1, 1
> y <- X %*% c(1,1,1) + rnorm(100)
```

Next, we maximize negative of sum of squared errors SSE (remember, `maxLik` is a maximizer not minimizer)

$$SSE(\boldsymbol{\beta}) = (\boldsymbol{y} - \mathsf{X}\cdot\boldsymbol{\beta})^\top(\boldsymbol{y} - \mathsf{X}\cdot\boldsymbol{\beta}) \tag{2}$$

as this is equivalent to likelihood maximization:

```
> negSSE <- function(beta) {
+    e <- y - X %*% beta
+    -crossprod(e)
+                             # note '-': we are maximizing
+ }
> m <- maxLik(negSSE, start=c(0,0,0))
>                             # give start values a bit off
> summary(m, eigentol=1e-15)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 1 iterations
Return code 3: Last step could not find a value above the current.
Boundary of parameter space?
Consider switching to a more robust optimisation method temporarily.
Log-Likelihood: -9.791126e+15
3  free parameters
Estimates:
       Estimate Std. error   t value Pr(> t)
[1,] -4.340e-01        NaN       NaN     NaN
[2,]  9.965e+03  3.397e-07 2.933e+10  <2e-16 ***
[3,]  9.765e-01  6.998e-09 1.395e+08  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

As one can see, the algorithm gets stuck and fails to converge, the last parameter value is also way off from the correct value $(1, 1, 1)$. We have amended summary with an extra argument, `eigentol=1e-15`. Otherwise `maxLik` refuses to compute standard errors for near-singular Hessian, see the documentation of `summary.maxLik`. It makes no difference right here but we want to keep it consistent with the two following examples.

Now let's improve the model performance with analytic gradient. The gradient of *SSE* can be written as

$$\frac{\partial}{\partial \boldsymbol{\beta}} SSE(\boldsymbol{\beta}) = -2(\boldsymbol{y} - \mathsf{X}\boldsymbol{\beta})^{\intercal}\mathsf{X}. \tag{3}$$

`maxLik` uses numerator layout, i.e. the derivative of the scalar log-likelihood with respect to the column vector of parameters is a row vector. We can code the negative of it as

```
> grad <- function(beta) {
+    2*t(y - X %*% beta) %*% X
+ }
```

We can add gradient to `maxLik` as an additional argument `grad`:

```
> m <- maxLik(negSSE, grad=grad, start=c(0,0,0))
> summary(m, eigentol=1e-15)
```

```
--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 3 iterations
Return code 2: successive function values within tolerance limit (tol)
Log-Likelihood: -94.77403
3  free parameters
Estimates:
      Estimate Std. error   t value Pr(> t)
[1,] 1.089e+00  7.727e-02 1.409e+01  <2e-16 ***
[2,] 1.000e+00  7.121e-05 1.404e+04  <2e-16 ***
[3,] 1.000e+00  7.000e-09 1.429e+08  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

Now the algorithm converges rapidly, and the estimate is close to the true value. Let us also add analytic Hessian, in this case it is

$$\frac{\partial^2}{\partial \boldsymbol{\beta}\, \partial \boldsymbol{\beta}^\intercal} SSE(\boldsymbol{\beta}) = 2\mathsf{X}^\intercal\mathsf{X} \tag{4}$$

and we implement the negative of it as

```
> hess <- function(beta) {
+    -2*crossprod(X)
+ }
```

Analytic Hessian matrix can be included with the argument `hess`, and now the results are

```
> m <- maxLik(negSSE, grad=grad, hess=hess, start=c(0,0,0))
> summary(m, eigentol=1e-15)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 3 iterations
Return code 2: successive function values within tolerance limit (tol)
Log-Likelihood: -94.77403
3  free parameters
Estimates:
      Estimate Std. error   t value Pr(> t)
[1,] 1.089e+00  7.728e-02 1.409e+01  <2e-16 ***
[2,] 1.000e+00  7.121e-05 1.404e+04  <2e-16 ***
[3,] 1.000e+00  7.000e-09 1.429e+08  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

Analytic Hessian did not change the convergence behavior here. Note that as the loss function is quadratic, Newton-Raphson should provide the correct

solution in a single iteration only. However, this example has numerical issues when inverting near-singular Hessian. One can easily check that when creating covariates in a less extreme scale, then the convergence is indeed immediate.

While using separate arguments `grad` and `hess` is perhaps the most straightforward way to supply gradients, `maxLik` also supports gradient and Hessian supplied as log-likelihood attributes. This is motivated by the fact that computing gradient often involves a number of similar computations as computing log-likelihood, and one may want to re-use some of the results. We demonstrate this on the same example, by writing a version of log-likelihood function that also computes the gradient and Hessian:

```
> negSSEA <- function(beta) {
+    ## negative SSE with attributes
+    e <- y - X %*% beta  # we will re-use 'e'
+    sse <- -crossprod(e)
+                            # note '-': we are maximizing
+    attr(sse, "gradient") <- 2*t(e) %*% X
+    attr(sse, "Hessian") <- -2*crossprod(X)
+    sse
+ }
> m <- maxLik(negSSEA, start=c(0,0,0))
> summary(m, eigentol=1e-15)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 3 iterations
Return code 2: successive function values within tolerance limit (tol)
Log-Likelihood: -94.77403
3  free parameters
Estimates:
      Estimate Std. error   t value Pr(> t)
[1,] 1.089e+00  7.727e-02 1.409e+01  <2e-16 ***
[2,] 1.000e+00  7.121e-05 1.404e+04  <2e-16 ***
[3,] 1.000e+00  7.000e-09 1.429e+08  <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
--------------------------------------------
```

The log-likelihood with “gradient” and “Hessian” attributes, `negSSEA`, computes log-likelihood as above, but also computes its gradient, and adds it as attribute “gradient” to the log-likelihood. This gives a potential efficiency gain as the residuals $\boldsymbol{e}$ are re-used. `maxLik` checks the presence of the attribute, and if it is there, it uses the provided gradient. In real applications the efficiency gain will depend on the amount of computations re-used, and the number of likelihood calls versus gradient calls.

While analytic gradients are always helpful and often necessary, they may be hard to derive and code. In order to help to derive and debug the analytic gradient, another provided function, `compareDerivatives`, takes the log-likelihood function, analytic gradient, and compares the numeric and analytic gradient.

As an example, we compare the log-likelihood and gradient functions we just coded:

```
> compareDerivatives(negSSE, grad, t0=c(0,0,0))

-------- compare derivatives --------
Function value:
[1] -1.021262e+16
Dim of analytic gradient: 1 3
       numeric          : 1 3
t0
     [,1] [,2] [,3]
[1,]    0    0    0
analytic gradient
         [,1]        [,2]        [,3]
[1,] 35754234 43262096385 2.04252e+16
numeric gradient
        [,1]       [,2]        [,3]
[1,] 3.8e+07 4.3262e+10 2.04252e+16
(anal-num)/(0.5*(abs(anal)+abs(num)))
            [,1]         [,2]         [,3]
[1,] -0.06089863 2.227923e-06 2.542083e-10
Max relative difference: 0.06089863
-------- END of compare derivatives --------

>                              # 't0' is the parameter value
```

The function prints the analytic gradient, numeric gradient, their relative difference, and the largest relative difference value (in absolute value). The latter is handy in case of large gradient vectors where it may be hard to spot a lonely component that is off. In case of reasonably smooth functions, expect the relative difference to be smaller than 10^{-7}. But in this example the numerical gradients are clearly problematic.

`compareDerivatives` supports vector functions, so one can test analytic Hessian in the same way by calling `compareDerivatives` with `gradlik` as the first argument and the analytic hessian as the second argument.

2.3 Different optimizers

By default, `maxLik` uses Newton-Raphson optimizer but one can easily swap the optimizer by `method` argument. The supported optimizers include "NR" for the default Newton-Raphson, "BFGS" for gradient-only Broyden-Fletcher-Goldfarb-Shannon, "BHHH" for the information-equality based Berndt-Hall-Hall-Hausman, and "NM" for gradient-less Nelder-Mead. Different optimizers may be based on a very different approach, and certain concepts, such as *iteration*, may mean quite different things.

For instance, although Newton-Raphson is a simple, fast and intuitive method that approximates the function with a parabola, it needs to know the Hessian matrix (the second derivatives). This is usually even harder to program than gradient, and even slower and more error-prone when computed numerically. Let us replace NR with gradient-only BFGS method. It is a quasi-Newton

method that computes its own internal approximation of the Hessian while relying only on gradients. We re-use the data and log-likelihood function from the first example where we estimated normal distribution parameters:

```
> m <- maxLik(loglik, start=c(mu=1, sigma=2),
+             method="BFGS")
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BFGS maximization, 20 iterations
Return code 0: successful convergence
Log-Likelihood: -144.6843
2  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.01021    0.10281  -0.099   0.921
sigma  1.02830    0.07271  14.143  <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
--------------------------------------------
```

One can see that the results were identical, but while NR converged in 7 iterations, it took 20 iterations for BFGS. In this example the BFGS approximation errors were larger than numeric errors when computing Hessian, but this may not be true for more complex objective functions. In a similar fashion, one can simply drop in most other provided optimizers.

One method that is very popular for ML estimation is BHHH. We discuss it here at length because that method requires both log-likelihood and gradient function to return a somewhat different value. The essence of BHHH is information equality, the fact that in case of log-likelihood function $\ell(\theta)$, the expected value of Hessian at the true parameter value $\boldsymbol{\theta}_0$ can be expressed through the expected value of the outer product of the gradient:

$$\mathbb{E}\left[\frac{\partial^2 l(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\,\partial\boldsymbol{\theta}^{\mathrm{T}}}\right]_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} = -\,\mathbb{E}\left[\left.\frac{\partial l(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}^{\mathrm{T}}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \cdot \left.\frac{\partial l(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0}\right]. \tag{5}$$

Hence we can approximate Hessian by the average outer product of the gradient. Obviously, this is only an approximation, and it is less correct when we are far from the true value $\boldsymbol{\theta}_0$. Note also that when approximating expected value with average we rely on the assumption that the observations are independent. This may not be true for certain type of data, such as time series.

However, in order to compute the average outer product, we need to compute gradient *by observation*. Hence it is not enough to just return a single gradient vector, we have to compute a matrix where rows correspond to individual data points and columns to the gradient components.

We demonstrate BHHH method by replicating the normal distribution example from above. Remember, the normal probability density is

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}}\frac{1}{\sigma}\,\mathrm{e}^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}. \tag{6}$$

and hence the log-likelihood contribution of x is

$$\ell(\mu, \sigma; x) = -\log\sqrt{2\pi} - \log\sigma - \frac{1}{2}\frac{(x-\mu)^2}{\sigma^2} \tag{7}$$

and its gradient

$$\begin{aligned} \frac{\partial}{\partial\mu}\ell(\mu, \sigma; x) &= \frac{1}{\sigma^2}(x-\mu) \\ \frac{\partial}{\partial\sigma}\ell(\mu, \sigma; x) &= -\frac{1}{\sigma} + \frac{1}{\sigma^2}(x-\mu)^2. \end{aligned} \tag{8}$$

We can code these two functions as

```
> loglik <- function(theta) {
+    mu <- theta[1]
+    sigma <- theta[2]
+    N <- length(x)
+    -N*log(sqrt(2*pi)) - N*log(sigma) - sum(0.5*(x - mu)^2/sigma^2)
+                              # sum over observations
+ }
> gradlikB <- function(theta) {
+    ## BHHH-compatible gradient
+    mu <- theta[1]
+    sigma <- theta[2]
+    N <- length(x)  # number of observations
+    gradient <- matrix(0, N, 2)  # gradient is matrix:
+                              # N datapoints (rows), 2 components
+    gradient[, 1] <- (x - mu)/sigma^2
+                              # first column: derivative wrt mu
+    gradient[, 2] <- -1/sigma + (x - mu)^2/sigma^3
+                              # second column: derivative wrt sigma
+    gradient
+ }
```

Note that in this case we do not sum over the individual values in the gradient function (but we still do in log-likelihood). Instead, we fill the rows of the $N \times 2$ gradient matrix with the values observation-wise.

The results are similar to what we got above and the convergence speed is in-between that of Newton-Raphson and BFGS:

```
> m <- maxLik(loglik, gradlikB, start=c(mu=1, sigma=2),
+             method="BHHH")
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BHHH maximisation, 16 iterations
Return code 8: successive function values within relative tolerance limit (reltol)
Log-Likelihood: -144.6843
2  free parameters
Estimates:
```

```
      Estimate Std. error t value Pr(> t)
mu    -0.01006    0.11046  -0.091   0.927
sigma  1.02816    0.08615  11.934  <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
--------------------------------------------
```

In case we do not have time and energy to code the analytic gradient, we can let `maxLik` compute the numeric one for BHHH too. In this case we have to supply the log-likelihood by observation. This essentially means we remove summing from the original likelihood function:

```
> loglikB <- function(theta) {
+    mu <- theta[1]
+    sigma <- theta[2]
+    -log(sqrt(2*pi)) - log(sigma) - 0.5*(x - mu)^2/sigma^2
+                            # no summing here
+                            # also no 'N*' terms as we work by
+                            # individual observations
+ }
> m <- maxLik(loglikB, start=c(mu=1, sigma=2),
+             method="BHHH")
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BHHH maximisation, 16 iterations
Return code 8: successive function values within relative tolerance limit (reltol)
Log-Likelihood: -144.6843
2  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.01006    0.11046  -0.091   0.927
sigma  1.02816    0.08615  11.934  <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
--------------------------------------------
```

Besides of relying on information equality, BHHH is essentially the same algorithm as NR. As the Hessian is just approximated, its is converging at a slower pace than NR with analytic Hessian. But when relying on numeric derivatives only, BHHH may be more reliable.

For convenience, the other methods also support observation-wise gradients and log-likelihood values, those numbers are just summed internally. So one can just code the problem in an BHHH-compatible manner and use it for all supported optimizers.

`maxLik` package also includes stochastic gradient ascent optimizer. As that method is rarely used for ML estimation, it cannot be supplied through the “method” argument. Consult the separate vignette “Stochastic gradient ascent in `maxLik`”.

2.4 Control options

`maxLik` supports a number of control options, most of which can be supplied through `control=list(...)` method. Some of the most important options include `printLevel` to control debugging information, `iterLim` to control the maximum number of iterations, and various `tol`-parameters to control the convergence tolerances. For instance, we can limit the iterations to two, while also printing out the parameter estimates at each step. We use the previous example with BHHH optimizer:

```
> m <- maxLik(loglikB, start=c(mu=1, sigma=2),
+             method="BHHH",
+             control=list(printLevel=3, iterlim=2))

Initial function value: -187.1825
Initial gradient value:
       mu     sigma
-25.25517 -24.02609
----- Initial parameters: -----
fcn value: -187.1825
      parameter initial gradient free
mu            1        -25.25517    1
sigma         2        -24.02609    1
Condition number of the (active) hessian: 1.213316
function value difference -2598.628 -> step 0.5
-----Iteration 1 -----
lambda  0  step 0.5  fcn value: -145.45843547
        amount  new param new gradient active
mu    1.805752 0.09712381    -9.103843      1
sigma 1.828402 1.08579876    -8.596451      1
Condition number of the hessian: 2.053022
-----Iteration 2 -----
lambda  0  step 1  fcn value: -144.69018329
          amount    new param new gradient active
mu    0.10199838 -0.004874569   -0.5111141      1
sigma 0.06440369  1.021395073    1.3303155      1
Condition number of the hessian: 3.15706
--------------
Iteration limit exceeded (iterlim)
2  iterations
estimate: -0.004874569 1.021395
Function value: -144.6902

> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BHHH maximisation, 2 iterations
Return code 4: Iteration limit exceeded (iterlim)
Log-Likelihood: -144.6902
2  free parameters
```

```
Estimates:
       Estimate Std. error t value Pr(> t)
mu    -0.004875   0.108699  -0.045   0.964
sigma  1.021395   0.084457  12.094  <2e-16 ***
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
--------------------------------------------
```

The first option, `printLevel=3`, make `maxLik` to print out parameters, gradient a few other bits of information at every step. Larger levels output more information, printlevel 1 only prints the first and last parameter values. The output from `maxLik`-implemented optimizers is fairly consistent, but methods that call optimizers in other packages, such as BFGS, may output debugging information in a quite different way. The second option, `iterLim=2` stops the algorithm after two iterations. It returns with code 4: iteration limit exceeded.

Other sets of handy options are the convergence tolerances. There are three convergence tolerances:

tol This measures the absolute convergence tolerance. Stop if successive function evaluations differ by less than *tol* (default 10^{-8}).

reltol This is somewhat similar to *tol*, but relative to the function value. Stop if successive function evaluations differ by less than $reltol \cdot (\ell(\boldsymbol{\theta}) + reltol)$ (default `sqrt(.Machine[["double.eps"]])`, may be approximately 1e-08 on a modern computer).

gradtol stop if the (Euclidean) norm of the gradient is smaller than this value (default 10^{-6}).

Default tolerance values are typically good enough, but in certain cases one may want to adjust these. For instance, in case of function values are very large, one may rely only on tolerance, and ignore relative tolerance and gradient tolerance criteria. A simple way to achieve this is to set both *reltol* and *gradtol* to zero. In that case these two conditions are never satisfied and the algorithm stops only when the absolute convergence criterion is fulfilled. For instance, in the previous case we get:

```
> m <- maxLik(loglikB, start=c(mu=1, sigma=2),
+             method="BHHH",
+             control=list(reltol=0, gradtol=0))
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
BHHH maximisation, 26 iterations
Return code 2: successive function values within tolerance limit (tol)
Log-Likelihood: -144.6843
2  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.01020    0.11050  -0.092   0.926
```

```
sigma  1.02829    0.08619  11.931  <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
--------------------------------------------
```

When comparing the result with that on Page 9 we can see that the optimizer now needs more iterations and it stops with a return code that is related to tolerance, not relative tolerance.

Note that BFGS and other optimizers that are based on the `stats::optim` does not report the convergence results in a similar way as BHHH and NR, the algorithms provided by the `maxLik` package. Instead of tolerance limits or gradient close to zero message, we hear about “successful convergence”. Stochastic gradient ascent relies on completely different convergence criteria. See the dedicated vignette “Stochastic Gradient Ascent in `maxLik`”.

3 Advanced usage

This section describes more advanced and less frequently used aspects of `maxLik`.

3.1 Additional arguments to the log-likelihood function

`maxLik` expects the first argument of log-likelihood function to be the parameter vector. But the function may have more arguments. Those can be passed as additional named arguments to `maxLik` function. For instance, let's change the log-likelihood function in a way that it expects data $\boldsymbol{x}$ to be passed as an argument `x`. Now we have to call `maxLik` with an additional argument `x=...`:

```
> loglik <- function(theta, x) {
+    mu <- theta[1]
+    sigma <- theta[2]
+    sum(dnorm(x, mean=mu, sd=sigma, log=TRUE))
+ }
> m <- maxLik(loglik, start=c(mu=1, sigma=2), x=x)
>                             # named argument 'x' will be passed
>                             # to loglik
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 7 iterations
Return code 1: gradient close to zero (gradtol)
Log-Likelihood: -144.6843
2  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.01021    0.10282  -0.099   0.921
sigma  1.02830    0.07271  14.142  <2e-16 ***
---
Signif. codes:
```

```
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
---------------------------------------------
```

This approach only works if the argument names do not overlap with `maxLik`'s arguments' names. If that happens, it prints an informative error message.

3.2 Maximizing other functions

`maxLik` function is basically a wrapper around a number of maximization algorithms, and a set of likelihood-related methods, such as standard errors. However, from time-to-time we need to optimize other functions where inverting the Hessian to compute standard errors is not applicable. In such cases one can call the included optimizers directly, using the form `maxXXX` where `XXX` stands for the name of the method, e.g. `maxNR` for Newton-Rapshon (`method="NR"`) and `maxBFGS` for BFGS. There is also `maxBHHH` although the information equality–based BHHH is not correct if we do not work with log-likelihood functions. The arguments for `maxXXX`-functions are largely similar to those for `maxLik`, the first argument is the function, and one also has to supply start values.

Let us demonstrate this functionality by optimizing 2-dimensional bell curve,

$$f(x, y) = \mathrm{e}^{-x^2-y^2}. \tag{9}$$

We code this function and just call `maxBFGS` on it:

```
> f <- function(theta) {
+    x <- theta[1]
+    y <- theta[2]
+    exp(-x^2 - y^2)
+                              # optimum at (0, 0)
+ }
> m <- maxBFGS(f, start=c(1,1))
>                             # give start value a bit off
> summary(m)

--------------------------------------------
BFGS maximization
Number of iterations: 9
Return code: 0
successful convergence
Function value: 1
Estimates:
          estimate     gradient
[1,] -1.212621e-09 2.442491e-09
[2,] -1.212621e-09 2.442491e-09
--------------------------------------------
```

Note that the summary output is slightly different: it reports the parameter and gradient value, appropriate for a task that is not likelihood optimization. Behind the scenes, this is because the `maxXXX`-functions return an object of *maxim*-class, not *maxLik*-class.

3.3 Testing condition numbers

Analytic gradient we demonstrated in Section 2.2 helps to avoid numerical problems. But not all problems can or should be solved by analytic gradients. For instance, multicollinearity should be addressed on data or model level. `maxLik` provides a helper function, `condiNumbers`, to detect such problems. We demonstrate this by creating a highly multicollinear dataset and estimating a linear regression model. We re-use the regression code from Section 2.2 but this time we create multicollinear data in similar scale.

```
> ## create 3 variables, two independent, third collinear
> x1 <- rnorm(100)
> x2 <- rnorm(100)
> x3 <- x1 + x2 + rnorm(100, sd=1e-6)  # highly correlated w/x1, x2
> X <- cbind(x1, x2, x3)
> y <- X %*% c(1, 1, 1) + rnorm(100)
> m <- maxLik(negSSEA, start=c(x1=0, x2=0, x3=0))
>                             # negSSEA: negative sum of squared errors
>                             # with gradient, hessian attribute
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 2 iterations
Return code 8: successive function values within relative tolerance limit (reltol)
Log-Likelihood: -92.66917
3  free parameters
Estimates:
   Estimate Std. error t value Pr(> t)
x1   0.7259        NaN     NaN     NaN
x2   0.3901        NaN     NaN     NaN
x3   1.5684        NaN     NaN     NaN
--------------------------------------------
```

As one can see, the model converges but the standard errors are missing (because Hessian is not negative definite).

In such case we may learn more about the problem by testing the condition numbers κ of either the design matrix X or of the Hessian matrix. It is instructive to test not just the whole matrix, but to do it column-by-column, and see where the number suddenly jumps. This hints which variable does not play nicely with the rest of data. `condiNumber` provides such functionality. First, we test the condition number of the design matrix:

```
> condiNumber(X)

x1          1
x2          1.353197
x3          5740575
```

We can see that when only including $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ into the design, the condition number is 1.35, far from any singularity-related problems. However, adding $\boldsymbol{x}_3$ to the matrix causes κ to jump to over 5 millions. This suggests that $\boldsymbol{x}_3$ is

highly collinear with $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. In this example the problem is obvious as this is how we created $\boldsymbol{x}_3$, in real applications one often needs further analysis. For instance, the problem may be in categorical values that contain too few observations or complex fixed effects that turn out to be perfectly multicollinear. A good suggestion is to estimate a linear regression model where one explains the offending variable using all the previous variables. In this example we might estimate `lm(x3 ~ x1 + x2)` and see which variables help to explain $\boldsymbol{x}_3$ perfectly.

Sometimes the design matrix is fine but the problem arises because data and model do not match. In that case it may be more informative to test condition number of Hessian matrix instead. The example below creates a linearly separated set of observations and estimates this with logistic regression. As a refresher, the log-likelihood of logistic regression is

$$\ell(\beta) = \sum_{i:y_i=1} \log \Lambda(\boldsymbol{x}_i^\mathsf{T}\boldsymbol{\beta}) + \sum_{i:y_i=0} \log \Lambda(-\boldsymbol{x}_i^\mathsf{T}\boldsymbol{\beta}) \tag{10}$$

where $\Lambda(x) = 1/(1+\exp(-x))$ is the logistic cumulative distribution function. We implement it using `R` function `plogis`

```
> x1 <- rnorm(100)
> x2 <- rnorm(100)
> x3 <- rnorm(100)
> X <- cbind(x1, x2, x3)
> y <- X %*% c(1, 1, 1) > 0
>                              # y values 1/0 linearly separated
> loglik <- function(beta) {
+    link <- X %*% beta
+    sum(ifelse(y > 0, plogis(link, log=TRUE),
+               plogis(-link, log=TRUE)))
+ }
> m <- maxLik(loglik, start=c(x1=0, x2=0, x3=0))
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 11 iterations
Return code 1: gradient close to zero (gradtol)
Log-Likelihood: -1.330027e-51
3  free parameters
Estimates:
    Estimate Std. error t value Pr(> t)
x1 9.046e+03        NaN     NaN     NaN
x2 9.063e+03  3.442e+25       0       1
x3 8.671e+03        NaN     NaN     NaN
--------------------------------------------
```

Not surprisingly, all coefficients tend to infinity and inference is problematic. In this case the design matrix does not show any issues:

```
> condiNumber(X)
```

```
x1          1
x2          1.14411
x3          1.229549
```

But the Hessian reveals that including $\boldsymbol{x}_3$ in the model is still problematic:

```
> condiNumber(hessian(m))
```

```
x1          1
x2          1.259299
x3          21.5091
```

Now the problem is not multicollinearity but the fact that $\boldsymbol{x}_3$ makes the data linearly separable. In such cases we may want to adjust our model or estimation strategy.

3.4 Fixed parameters and constrained optimization

`maxLik` supports three types of constrains. The simplest case just keeps certain parameters' values fixed. The other two, general linear equality and inequality constraints are somewhat more complex.

Occasionally we want to treat one of the model parameters as constant. This can be achieved in a very simple manner, just through the argument `fixed`. It must be an index vector, either numeric, such as `c(2,4)`, logical as `c(FALSE, TRUE, FALSE, TRUE)`, or character as `c("beta2", "beta4")` given `start` is a named vector. We revisit the first example of this vignette and estimate the normal distribution parameters again. However, this time we fix $\sigma = 1$:

```
> x <- rnorm(100)
> loglik <- function(theta) {
+    mu <- theta[1]
+    sigma <- theta[2]
+    sum(dnorm(x, mean=mu, sd=sigma, log=TRUE))
+ }
> m <- maxLik(loglik, start=c(mu=1, sigma=1),
+             fixed="sigma")
>                             # fix the component named 'sigma'
> summary(m)

--------------------------------------------
Maximum Likelihood estimation
Newton-Raphson maximisation, 2 iterations
Return code 8: successive function values within relative tolerance limit (reltol)
Log-Likelihood: -140.4475
1  free parameters
Estimates:
      Estimate Std. error t value Pr(> t)
mu    -0.11337    0.09999  -1.134   0.257
sigma  1.00000    0.00000      NA      NA
--------------------------------------------
```

The result has σ exactly equal to 1, it's standard error 0, and t value undefined. The fixed components are ignored when computing gradients and Hessian in the optimizer, essentially reducing the problem from 2-dimensional to 1-dimensional. Hence the inference for μ is still correct.

Next, we demonstrate equality constraints. We take the two-dimensional function we used in Section 3.2 and add constraints $x + y = 1$. The constraint must be described in matrix form $\mathsf{A}\boldsymbol{\theta} + \boldsymbol{B} = 0$ where $\boldsymbol{\theta}$ is the parameter vector and matrix A and vector $\boldsymbol{B}$ describe the constraints. In this case we can write

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -1 \end{pmatrix} = 0, \tag{11}$$

i.e. $\mathsf{A} = (1\ 1)$ and $\boldsymbol{B} = -1$. These values must be supplied to the optimizer argument `constraints`. This is a list with components names `eqA` and `eqB` for A and $\boldsymbol{B}$ accordingly. We do not demonstrate this with a likelihood example as no corrections to the Hessian matrix is done and hence the standard errors are incorrect. But if you are not interested in likelihood-based inference, it works well:

```
> f <- function(theta) {
+    x <- theta[1]
+    y <- theta[2]
+    exp(-x^2 - y^2)
+                              # optimum at (0, 0)
+ }
> A <- matrix(c(1, 1), ncol=2)
> B <- -1
> m <- maxNR(f, start=c(1,1),
+            constraints=list(eqA=A, eqB=B))
> summary(m)

--------------------------------------------
Newton-Raphson maximisation
Number of iterations: 1
Return code: 1
gradient close to zero (gradtol)
Function value: 0.6065399
Estimates:
      estimate   gradient
[1,] 0.4999848 -0.6065307
[2,] 0.4999848 -0.6065307

Constrained optimization based on SUMT
Return code: 1
penalty close to zero
5  outer iterations, barrier value 9.196982e-10
--------------------------------------------
```

The problem is solved using sequential unconstrained maximization technique (SUMT). The idea is to add a small penalty for the constraint violation, and to slowly increase the penalty until violations are prohibitively expensive. As the example indicates, the solution is extremely close to the constraint line.

The usage of inequality constraints is fairly similar. We have to code the inequalities as $\mathsf{A}\boldsymbol{\theta} + \boldsymbol{B} > 0$ where the matrices A and $\boldsymbol{B}$ are defined as above. Let us optimize the function over the region $x + y > 1$. In matrix form this will be

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + (-1) > 0. \tag{12}$$

Supplying the constraints is otherwise similar to the equality constraints, just the constraints-list components must be called `ineqA` and `ineqB`. As `maxNR` does not support inequality constraints, we use `maxBFGS` instead. The corresponding code is

```
> A <- matrix(c(1, 1), ncol=2)
> B <- -1
> m <- maxBFGS(f, start=c(1,1),
+              constraints=list(ineqA=A, ineqB=B))
> summary(m)

--------------------------------------------
BFGS maximization
Number of iterations: 42
Return code: 0
successful convergence
Function value: 0.6064307
Estimates:
      estimate   gradient
[1,] 0.5000824 -0.6065306
[2,] 0.5000824 -0.6065306

Constrained optimization based on constrOptim
1  outer iterations, barrier value -0.0009710671
--------------------------------------------
```

Not surprisingly, the result is exactly the same as in case of equality constraints, in this case the optimum is found at the boundary line, the same line what we specified when demonstrating the equality constraints.

One can supply more than one set of constraints, in that case these all must be satisfied at the same time. For instance, let's add another condition, $x - y > 1$. This should be coded as another line of A and another component of $\boldsymbol{B}$, in matrix form the constraint is now

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} > \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{13}$$

where "$>$" must be understood as element-wise operation. We also have to ensure the initial value satisfies the constraint, so we choose $\boldsymbol{\theta}_0 = (2, 0)$. The code will be accordingly:

```
> A <- matrix(c(1, 1, 1, -1), ncol=2)
> B <- c(-1, -1)
> m <- maxBFGS(f, start=c(2, 0),
+              constraints=list(ineqA=A, ineqB=B))
> summary(m)
```

```
--------------------------------------------
BFGS maximization
Number of iterations: 56
Return code: 0
successful convergence
Function value: 0.3676795
Estimates:
     estimate   gradient
[1,] 1.000272 -0.7355588
[2,] 0.000000  0.0000000

Constrained optimization based on constrOptim
1  outer iterations, barrier value -0.00184214
--------------------------------------------
```

The solution is $(1, 0)$ the closest point to the origin where both constraints are satisfied.

This example concludes the `maxLik` usage introduction. For more information, consult the fairly extensive documentation, and the other vignettes.