Generalized nonlinear models in R: an overview of the `gnm` package

Heather Turner and David Firth

May 13, 2005

Contents

1 Introduction

```
> library(gnm)
```

2 Generalized Linear Models

3 Nonlinear Terms

The `gnm` package provides a flexible framework for the specification and estimation of generalized models with nonlinear terms. Multiplicative interaction terms can be estimated using the in-built capability of the `gnm` function and are specified in the model formula using the symbolic function `Mult`. Other nonlinear terms can be estimated using plug-in functions for `gnm` and are specified using `Nonlin`.

There are two plug-in functions currently available in the `gnm` package: `MultHomog` for fitting multiplicative interaction terms with homogeneous effects and `Dref` for fitting diagonal reference terms. Users may also define custom plug-in functions to fit other types of nonlinear terms.

3.1 Multiplicative Interaction Terms using `Mult`

Multiplicative interaction terms can be included in the formula argument to `gnm` by using the symbolic wrapper function `Mult`. Factors in the interaction are passed as unspecified arguments to `Mult` and are expressed by symbolic linear formulae. An intercept is automatically added to each factor unless otherwise specified. For example, to fit the row-column association model

$$\log \mu_{rc} = \alpha_r + \beta_c + \gamma_r \delta_c,$$

also known as the Goodman RC model [2], the `formula` argument of `gnm` would be

```
mu ~ R + C + Mult(-1 + R, -1 + C)
```

where R and C are row and column factors respectively.

`Mult` has one specified argument `multiplicity`, which is `1` by default. This argument determines the number of multiplicative components that are fitted. For example,

```
mu ~ R + C + Mult(-1 + R, -1 + C, multiplicity = 2)
```

would give the RC(2) model [2]

$$\log \mu_{rc} = \alpha_r + \beta_c + \gamma_r \delta_c + \theta_r \phi_c.$$

In some contexts, it may be desirable to constrain one or more of the multiplicative factors so that the factor is always nonnegative. This may be achieved by defining the factor as an exponential, as in the following 'uniform difference' model [3, 1]

$$\log \mu_{ijt} = \alpha_{it} + \beta_{jt} + e^{\gamma_t} \delta_{ij}.$$

Exponentiated factors can be specified in `gnm` models using the symbolic function `Exp`, for example the uniform difference model above would be specified by the formula

```
mu ~ R:T + C:T + Mult(Exp(-1 + T), R:C, multiplicity = 2)
```

3.2 Other Nonlinear Terms using `Nonlin`

Nonlinear terms which can not be specified using `Mult` may be specified using `Nonlin`. This symbolic function indicates a term which requires a plug-in function to estimate the associated parameters. There are two arguments to `Nonlin`: a call to the relevant plug-in

function and if necessary, a data.frame containing any variables that are required by specified arguments of the plug-in function, which do not appear in any unspecified arguments of the plug-in function or elsewhere in the model formula.

For example, in the formula

```
mu ~ x + A + B + Nonlin(PlugInFunction(A, B, arg1 = x, arg2 = C),
                        data = data.frame.of.C)
```

`Nonlin` is used to specify a term that requires the plug-in function PlugInFunction. As the factor C only appears in the specified arguments of the call to PlugInFunction, a data.frame containing factor C has been passed to the `data` argument of `Nonlin`. Note that this would not be necessary if C could be found in an environment on the search path (given by `search()`).

The two plug-in functions included in the `gnm` package are described below, followed by a guide to writing custom plug-in functions.

3.2.1 `MultHomog`

The `MultHomog` function provides the tools required to fit multiplicative interaction terms in which the level effects are constrained to be equal across the factors. The arguments of `MultHomog` are the factors in the interaction, which are assumed to be objects of class "factor". Like a `Mult` term, the interaction can include any number of factors, but there is no multiplicity argument.

As an example, consider the following association model with homogeneous row-column effects

$$\log \mu_{rc} = \alpha_r + \beta_c + \theta_{rc} + \gamma_r \gamma_c.$$

To fit this model, the formula argument to `gnm` would be

```
mu ~ R + C + Diag(R, C) + Nonlin(MultHomog(R, C))
```

If the factors passed to `MultHomog` do not have exactly the same levels, a common set of levels is obtained by taking the union of the levels of each factor, sorted into increasing order.

3.2.2 `Dref`

`Dref` is a plug-in function to fit diagonal reference terms involving two or more factors with a common set of levels. A diagonal reference term comprises an additive component for each factor. For a given data point, the component for the i'th factor, say F, is

$$w_i \gamma_f$$

where w_i is the weight for factor i, γ_f is the "diagonal effect" for level f and f is the level of F for the given data point.

The weights are constrained to be nonnegative and to sum to one so that a "diagonal effect", say γ_l, is the value of the diagonal reference term for data points with level l across the factors. `Dref` constrains the weights by defining them as

$$w_i = \frac{e^{\delta_i}}{\sum_r^n e^{\delta_r}}$$

and estimating the δ_i.

Factors in the interaction are passed to unspecified arguments of `Dref`. For example, the following diagonal reference model for a contingency table classified by the row factor R and the column factor C

$$\mu_{rc} = \frac{e^{\delta_1}}{e^{\delta_1} + e^{\delta_2}}\gamma_r + \frac{e^{\delta_2}}{e^{\delta_1} + e^{\delta_2}}\gamma_c,$$

would be specified by the formula

```
mu ~ -1 + Nonlin(Dref(R, C))
```

`Dref` has one specified argument `formula`, which is a symbolic description of the dependence of δ_i on any covariates. For example, the formula

```
mu ~ -1 + x + Nonlin(Dref(R, C, formula = ~ 1 + x))
```

specifies the following diagonal reference model

$$\mu_{rc} = \beta_X x + \frac{e^{\xi_1+\beta_1 x}}{e^{\xi_1+\beta_1 x} + e^{\xi_2+\beta_2 x}}\gamma_r + \frac{e^{\xi_2+\beta_2 x}}{e^{\xi_1+\beta_x} + e^{\xi_2+\beta_2 x}}\gamma_c,$$

The default value of `formula` is $\sim$ `1`, so that constant weights are estimated. The coefficients returned by `gnm` are those that are directly estimated, i.e. the δ_i or the ξ_i and β_i, rather than the implied weights w_i.

3.2.3 Custom Plug-in Functions

Custom plug-in functions may be written to enable `gnm` to fit nonlinear terms that can not be specifed by `Mult` or the plug-in functions provided by the `gnm` package.

There are no constraints on the arguments that a plug-in function may have. However it should not be assumed that model variables exist in an environment on the search path, since `gnm` does not assume this. Rather the function `getModelFrame` should be used to get the model.frame used by `gnm`, which will have all the model variables and also attributes useful for model.matrix etc.

For example, the first few lines of the `MultHomog` function are

```
MultHomog <- function(...){
  labelList <- as.character((match.call(expand.dots = FALSE))[[2]])
  gnmData <- getModelFrame()
  designList <- lapply(gnmData[, labelList], class.ind)
...
```

The names of the factors in the interaction are assigned to `labelList`, and the model.frame used by `gnm` is assigned to `gnmData`. The factors can then be accessed by name from `gnmData`, as in the call to `lapply`.

The plug-in function should return a list with the following components

start (optional) either a vector of default starting values for the parameters or a function which takes the number of parameters and returns a vector of default starting values. See Section 4.2 for details of how these values will be used if provided and the generic default values that will be used otherwise.

labels a character vector of labels for the parameters (to which `gnm` will prefix the call to the plug-in function).

predictor a function which takes a vector of parameter estimates and returns either a vector of fitted values or a matrix whose columns are additive components of the fitted values.

localDesignFunction a function which takes the specified arguments `coef` (a vector of parameter estimates) and `predictor` (the result of the predictor function), and returns the local design matrix.

As an example of a `start` component, `Dref` simply returns

```
rep(0.5, length(labels))
```

where `labels` is the vector of parameter labels to be returned as the `labels` component, for instance

```
c("A", "B", "1", "2", "3", "4", "5", "6", "7")
```

The `MultHomog` function provides a simple example of a `predictor` component:

```
predictor <- function(coef) {
    do.call("pprod", lapply(designList, "%*%", coef))
}
```

which computes the product of the vectors found by multiplying the design matrix for each factor in the interaction (held in `designList`) by the homogeneous coefficients (in `coef`). This function takes advantage of *lexical scoping*: `designList` is an object defined in `MultHomog`, which `predictor` is able to find because `predictor` is also defined in `MultHomog` and hence `MultHomog` is the enclosing environment of `predictor`.

The `localDesignFunction` created by `MultHomog` is slightly more complicated:

```
localDesignFunction <- function(coef, ...) {
    productList <- designList
    for (i in seq(designList))
        productList[[i]] <- designList[[i]] *
            drop(do.call("pprod", lapply(designList[-i], "%*%", coef)))
    do.call("psum", productList)
}
```

This function only requires the argument `coef`, but since the local design function returned by a plug-in function must also take the argument `predictor`, further arguments are allowed by the use of the special argument "...".

4 Controlling the Fitting Procedure

`gnm` has a number of arguments which affect the way a model will be fitted. Basic control parameters and starting values can be set by `control` and `start` respectively. Parameters can be constrained to zero by specifying a `constrain` argument. Finally parameters of a stratification factor can be handled more efficiently by specifying the term in an `eliminate` argument. These options are described in more detail below.

4.1 Using `control` with `gnmControl`

The `control` argument provides a way to specify the tolerance level for convergence, the number of starting iterations and the maximum number of main iterations, as well as the option to trace the deviance throughout the fitting process. By default, the `control` argument is a call to `gnmControl` using any arguments passed on from `gnm`. `gnmControl` creates a list of the control parameters, including any at their default values. For example

```
gnm(mu ~ R + C + Mult(-1 + R, -1 + C), tolerance = 1e-6, iterStart = 3)
```

is equivalent to

```
gnm(mu ~ R + C + Mult(-1 + R, -1 + C),
    control = gnmControl(tolerance = 1e-6, iterStart = 3))
```

which is the same as

```
gnm(mu ~ R + C + Mult(-1 + R, -1 + C),
    control = list(tolerance = 1e-6, iterStart = 3, iterMax = 500, trace = FALSE))
```

4.2 Using `start`

In some contexts, the default starting values may not be appropriate and the algorithm will fail to converge, or perhaps only converge after a large number of iterations. Alternative starting values may be passed on to `gnm` by specifying a `start` argument. This should be a numeric vector of length equal to the number of parameters (or possibly the non-eliminated parameters, see Section 4.4), however missing starting values (`NA`s) are allowed.

If there is no user-specified starting value for a parameter, the default value is used. This feature is particularly useful when adding terms to a model, since the estimates from the original model can be used as starting values, as in the example below

```
model1 <- gnm(mu ~ R + C + Mult(-1 + R, -1 + C))
model2 <- gnm(mu ~ R + C + Mult(-1 + R, -1 + C, multiplicity = 2),
              start = c(coef(model1), rep(NA, 10))
```

`gnm` can be run with `method = "coef"` to identify the parameters of a model prior to estimation, to assist with the specification of arguments such as `start`.

The starting procedure used by `gnm` is as follows

1. Generate starting values θ_i for all parameters $i = 1, \ldots, p$ from the Uniform(-0.1, 0.1) distribution. Shift these values away from zero as follows

$$\theta_i = \begin{cases} \theta_i - 0.1 & \text{if } \theta < 1 \\ \theta_i + 0.1 & \text{otherwise} \end{cases}$$

2. Replace generic starting values with any starting values specified by plug-in functions.

3. Replace default starting values with any starting values specified by the `start` argument of `gnm`.

4. Compute the `glm` estimate of any parameters in linear terms that were not specified by `start`, offsetting the contribution to the predictor of any parameters specified by `start` or a plug-in function.

5. Run starting iterations: update any parameters in nonlinear terms that were not specified by `start` or a plug-in function one at a time, updating *all* linear terms after each round of nonlinear updates.

Note that no starting iterations will be run if all parameters are specified by the `start` argument of `gnm`.

4.3 Using `constrain`

By default, `gnm` only imposes identifiability constraints on any linear terms in the model to be fitted. For these terms, the constraints are determined in the same way as they would be in `glm`. Any nonlinear terms will be over-parameterised unless constraints are imposed by the defining plug-in function (as in the case of `Dref` for example). For a model with nonlinear terms that are over-parameterised, `gnm` will return a random parameterisation.

To illustrate this point, consider the following application of `gnm`, discussed later in Section 6.1

```
> data(occupationalStatus, package = "gnm")
> set.seed(1)
> RChomog1 <- gnm(Freq ~ origin + destination + Diag(origin, destination) +
+     Nonlin(MultHomog(origin, destination)), family = poisson,
+     data = occupationalStatus)
```

Running the analysis again from a different seed

```
> set.seed(2)
> RChomog2 <- eval(RChomog1$call)
```

gives a different representation of the same model

```
> compareCoef <- cbind(coef(RChomog1), coef(RChomog2))
> colnames(compareCoef) <- c("RChomog1", "RChomog2")
> compareCoef
```

```
                                  RChomog1    RChomog2
(Intercept)                     0.01031358  0.10631042
origin2                         0.52684390  0.51997443
origin3                         1.65525382  1.62956305
origin4                         1.99636593  1.95230159
origin5                         0.77767542  0.73307058
origin6                         2.85898522  2.79827815
origin7                         1.54820728  1.47440621
origin8                         1.29563149  1.21416423
destination2                    0.94585703  0.93898798
destination3                    1.99966968  1.97397893
destination4                    2.28479944  2.24073545
destination5                    1.67709218  1.63248789
destination6                    3.16246317  3.10175638
destination7                    2.29980341  2.22600286
destination8                    1.87100856  1.78954180
Diag(origin, destination)1      1.52666556  1.52666846
Diag(origin, destination)2      0.45600920  0.45600795
```

```
Diag(origin, destination)3       -0.01597343 -0.01598066
Diag(origin, destination)4        0.38918303  0.38918427
Diag(origin, destination)5        0.73851492  0.73851696
Diag(origin, destination)6        0.13474284  0.13474352
Diag(origin, destination)7        0.45763249  0.45763821
Diag(origin, destination)8        0.38847753  0.38846397
MultHomog(origin, destination).1 -1.54111773 -1.50965033
MultHomog(origin, destination).2 -1.32282516 -1.29135537
MultHomog(origin, destination).3 -0.72465413 -0.69319228
MultHomog(origin, destination).4 -0.14077778 -0.10930985
MultHomog(origin, destination).5 -0.12361117 -0.09214108
MultHomog(origin, destination).6  0.38814928  0.41961438
MultHomog(origin, destination).7  0.80429340  0.83575531
MultHomog(origin, destination).8  1.04785874  1.07933252
```

```
> multCoef <- coef(RChomog1)[grep("Mult", names(coef(RChomog1)))]
> set.seed(1)
> RChomogConstrained1 <- update(RChomog1, constrain = length(coef(RChomog1)),
+     start = c(rep(NA, 23), multCoef - multCoef[8]))
> set.seed(2)
> RChomogConstrained2 <- eval(RChomogConstrained1$call)
> identical(coef(RChomogConstrained1), coef(RChomogConstrained2))
```

```
[1] TRUE
```

4.4 Using `eliminate`

5 Methods and Accessor functions

6 Examples

6.1 Row-column Association Models

6.2 Uniform Difference (UNIDIFF) Models

6.3 Generalized Additive and Mulipliactive (GAMMI) Models

6.4 Stereotype Models

A User-level Functions

References

[1] R Erikson and J H Goldthorpe. *The Constant Flux*. Oxford: Clarendon Press, 1992.

[2] L A Goodman. Simple models for the analysis of association in cross-classifications having ordered categories. *J. Amer. Statist. Assoc.*, 74:537–552, 1979.

[3] Y Xie. The log-multiplicative layer effect model for comparing mobility tables. *American Sociological Review*, 57:380–395, 1992.