Modern Data Analysis

in

S-PLUS

Workshop on 7 October, 1998
International S-PLUS User Conference
Washington, DC

Presenters:

Brian Ripley,
University of Oxford,
Oxford, UK.

Bill Venables,
University of Adelaide,
South Australia.

`ripley@stats.ox.ac.uk`
`http://www.stats.ox.ac.uk/~ripley`

`Bill.Venables@adelaide.edu.au`

Presentation 3

1. Structure of model fitting functions.

2. `terms` objects.

3. Designing a new model fitting function, an example: log-linear models.

The modern model fitting and Trellis graphics functions have several features in common. They *usually* provide

- A *formula* argument to specify the essential variables on which the operation depends,
- A *data frame* argument to provide a preferential frame for variables.
- Some conventional other arguments such as `subset`, `weights` and `na.action`.

The variables occurring in `formula`, `subset` and `weights` arguments may only be visible within the data frame. If they are visible in the data frame, that copy is always used.

The result of a model fitting operation is always an object with a suitably identifying *class* (normally the name of the fitting function).

Further operations (updating, printing, plotting, summarising, extracting coefficients, fitted values or residuals, making predictions, &c) are done by method functions.

(In the case of Trellis functions, *printing* the object is the only way to produce a plot.)

The formula for a model fitting function may be interpreted in any way the user requires. The operators have no intrinsic meaning. Some names, (e.g. '`.`' ('dot')), may have special meanings. 'Special' functions (`Error`, `strata`, `offset`, ...) are treated specially.

The binary operators include $|$ and `%in%` as well as the arithmetic operators. The model fitting function may impose restrictions on the kinds of formulae acceptable. (e.g. `tree` formulae may only contain $+$ operators, not even $-$)

The *linear model* group of model fitting functions (`lm`, `aov`, `glm`, `gam`, `manova`) interpret the formula as specifying a *model matrix*.

Formulae in these cases are processed into `terms` objects to give the characteristics of the matrix. (`gam` formulae allow some special extensions).

The function `terms` provides this parsing facility.

Model functions often construct a *frame* consisting of all variables needed for the operation.

In the case of the linear model group, this frame (called a *model frame*) also contains the terms object from the parsed formula.

Constructing a *model frame*

Most fitting functions construct the model frame by modifying the original call and re-presenting it to the evaluator in the parent frame.

The initial part of `lm` shows the usual idiom:

```
> lm
function(formula, data, weights, subset,
        na.action, method = "qr", model = F,
        x = F, y = F, contrasts = NULL, ...)
{
        call <- match.call()
        m <- match.call(expand = F)
        m$method <- m$model <- m$x <- m$y <-
                m$contrasts <- m$... <- NULL
        m[[1]] <- as.name("model.frame")
        m <- eval(m, sys.parent())
        if(method == "model.frame")
                return(m)
        Terms <- attr(m, "terms")
        weights <- model.extract(m, weights)
        Y <- model.extract(m, response)
        X <- model.matrix(Terms, m, contrasts)
....
```

Notice how extracting key quantities from the model frame is quite simple.

The remainder of `lm` is typical of how a model fitting function does its work.

```
....
  fit <- if(length(weights))
    lm.wfit(X, Y, weights, method, ...) else
    lm.fit(X, Y, method, ...)
  fit$terms <- Terms
  fit$call <- call
  if(model)
    fit$model <- m
  if(x)
    fit$x <- X
  if(y)
    fit$y <- Y
  attr(fit, "na.message") <-
      attr(m, "na.message")
  fit
}
```

After the call to `lm.fit` or `lm.wfit` the object has class `"lm"`. The rest of the function is ensuring that the object contains all requested components.

Anatomy of a fitting function - warts and all!

```
Brett Presnell's "literate" annotation of the glm() function.
          (Quoted here with permission, October, 1998)

-----------------------------------------------------------------

"glm" <- function(formula = formula(data), family = gaussian,
           data = sys.parent(), weights, subset, na.action,
           start = eta, control = glm.control(...),
           method = "glm.fit", model = F, x = F, y = T,
           contrasts = NULL, ...)
{
  ###############################################################
  ## THE FIRST SECTION IS FAIRLY GENERIC S MODELING STUFF
  ###############################################################

  ## The call is saved to be returned as part of the output.

  call <- match.call()

  ## Save the call again, but this time for internal purposes.
  ## We don't expand the dots because whatever is in there will
  ## not be used in the when we create the model frame below.

  m <- match.call(expand = F)

  ## The next line nullifies some arguments before evaluating
  ## model.frame on the remaining arguments (if they are
  ## specified in the call): formula,data,weights, subset,
  ## na.action, and start.  Did I miss anything?

  ## Note that match.call returns an object with mode "call",
  ## which is something like a list with first component being
  ## the function name.  The remaining components are the
  ## arguments actually specified in the function call.  In
  ## particular, if an argument has default values and the call
  ## doesn't override it, then match.call doesn't doesn't return
  ## it as part of its value.
```

```
## BEGIN WHINGING ABOUT OBSCURE CODE
## This behaviour of match.call, coupled with lazy evaluation,
## explains why the author(s) of this function can get away
## with "start=eta" whether eta exists or not.  Note that even
## if eta exists as an object in the parent frame it is ignored
## unless the user has specified "start=eta" explicitly in the
## call to glm.  This seems unnecessarily obtuse.  Why not just
## have "start=NULL" to avoid any possible confusion?  The only
## reason I can see for "start=eta" is as a sort of internal
## documentation, in that it indicates that "start" represents
## starting values for the linear predictor, but this of course
## assumes familiarity with the notation of say McCullagh and
## Nelder.
## END WHINGING

m$family <- m$method <- m$model <- m$x <- m$y <- m$control <-
  m$contrasts <- m$... <- NULL

## Replace "glm" with "model.frame" what is left of the call.

m[[1]] <- as.name("model.frame")

## Now evaluate m to create the data frame for further work.

m <- eval(m, sys.parent())

## This extracts and save the terms attribute, which is later
## used to create the model matrix.

Terms <- attr(m, "terms")

## It seems that the next line should logically go before the
## previous one, since here we just return the frame, in which
## case Terms is not used.

if(method == "model.frame")
  return(m)
```



```
## Interesting.  The attributes of the model frame are
## extracted and stored in "a", but "a" never seems to be used.
## Am I missing anything here?

a <- attributes(m)

## Extract the response, the X matrix, and the weights from the
## model frame.

Y <- model.extract(m, response)
X <- model.matrix(Terms, m, contrasts)
w <- model.extract(m, weights)

## If no weights are specified, set all weights to 1.  Also
## issue an error message and stop in case of any negative
## weights.

if(!length(w))
  w <- rep(1, nrow(m))
else if(any(w < 0))
  stop("negative weights not allowed")

## Extract the starting values for the linear predictor.

start <- model.extract(m, start)

## Extract the offset, which must be specified in the model
## formula.  Note that this is just a term who's regression
## coefficient is known (or assumed) to be 1.

offset <- model.extract(m, offset)
```

```
  ############################################################
  ## THIS SECTION IS MORE GLM SPECIFIC
  ############################################################

  ## Get the distributional family of the response and the
  ## fitting method to be used.

  family <- as.family(family)
  if(missing(method))
    method <- attr(family, "method")
  if(!is.null(method)) {
    if(!exists(method, mode = "function"))
      stop(paste("unimplemented method:", method))
  }
  else method <- "glm.fit"
  glm.fitter <- get(method)

  ## Compute the fit by calling a more bare bones fitting routine
  ## where the real computations are done.

  fit <- glm.fitter(x = X, y = Y, w = w, start = start,
             offset = offset, family = family,
             maxit = control$maxit,
             epsilon = control$epsilon,
             trace = control$trace, null.dev = T, ...)

  ## If an offset and intercept is present, iterations are needed
  ## to compute the Null deviance; these are done here, unless
  ## the model is NULL, in which case the computations have been
  ## done already

  if(any(offset) && attr(Terms, "intercept")) {
    null.deviance <-
      if(length(Terms))
        glm.fitter(X[,"(Intercept)", drop = F], Y,
                   w, offset = offset, family = family,
                   maxit = control$maxit,
                   epsilon = control$epsilon,
                   null.dev = NULL)$deviance
      else fit$deviance
    fit$null.deviance <- null.deviance
  }
```



```
  ##############################################################
  ## THE LAST SECTION IS AGAIN FAIRLY GENERIC
  ##############################################################

  ## Attach other desired information about the fit before
  ## returning.

  attr(fit, "class") <- c("glm", "lm")
  fit$terms <- Terms
  fit$formula <- as.vector(attr(Terms, "formula"))
  fit$call <- call
  if(model)
    fit$model <- m
  if(x)
    fit$x <- X
  if(!y)
    fit$y <- NULL
  fit
}
```

Changes needed for S4:

Replace

```
attr(fit, "class") <- c("glm", "lm")
```

by

```
oldClass(fit) <- "glm"
```

You also need to do
`setOldClass(c("glm", "lm"))`
somewhere to establish that objects of (old) class
`"glm"` inherit from (old) `"lm"`.

Building a new model function: An example

S-PLUS contains an old style fitting function, `loglin`, for fitting log-linear models to frequency arrays by iterative proportional scaling.

For large frequency arrays this is by far the most efficient method. The alternative—surrogate Poisson models—requires a typically huge model matrix to be calculated and stored. (Multinomial modelling provides a good compromise.)

The main arguments to `loglin` are

1. A regular *array* containing the frequencies,

2. A *list* specifying in a conventional (and awkward) way which faces of fitted and observed frequency arrays must have the same totals.

3. A *vector* of starting values, needed only if the model contains structural zeros.

Example of `loglin`

The array `barley.exposed` is an $8 \times 5 \times 3$ array giving the number of healthy young barley plants exposed to a disease, classified by *cultivar*, *time* period and *cluster* in the field.

We fit a formal log-linear model

```
> dim(barley.exposed)
[1] 8 5 3
> names(dimnames(barley.exposed))
[1] "cultivar" "time"     "cluster"
> mod1 <- loglin(barley.exposed,
        list(1:2, c(1,3), 2:3), fit = T)
4 iterations: deviation 0.0067215
> names(mod1)
[1] "lrt"    "pearson" "df"    "margin"  "fit"
> c(mod1$lrt, mod1$df)
[1] 93.959 56.000
```

We would like to be able to fit this model, using e.g.

```
fm = loglm(~(1 + 2 + 3)^2, barley.exposed)
fm2 = loglm(~1:2 + 1:3 + 2:3, barley.exposed)
```

and extract residuals, fitted values, perform tests, &c, in a way similar to that used by the linear model group.

Pre-parsing formulae.

For `terms` to parse formulae the names must be syntactically valid names.

```
denumerate = function(x)
  UseMethod("denumerate")

denumerate.formula = function(x) {
  if(length(x) == 1) {
    if(mode(x) == "numeric" ||
      (mode(x) == "name" &&
        any(substring(x, 1, 1) ==
          as.character(1:9))))
       x <- as.name(paste(".v", x, sep = ""))
    } else {
      x[[2]] <- Recall(x[[2]])
      if(length(x)==3 && x[[1]] !=as.name("^"))
      x[[3]] <- Recall(x[[3]])
    }
  x
}
```

Example:

```
> denumerate(~1:2 + 2:3 + 3:1)
 ~ .v1:.v2 + .v2:.v3 + .v3:.v1
> denumerate(~1:2 + 2:3 + A/B)
 ~ .v1:.v2 + .v2:.v3 + A/B
```

The ‘denumerated’ form can be parsed by `terms`.

`loglm` design considerations

1. Allow (formula, array) entry, in which case the formula may be in dimension number or name form. The array holds the frequencies.
2. Allow (formula, crosstabs-object) entry, in which case the same protocol applies as for the previous case.
3. Allow (formula, data.frame) entry, in which case the left hand side of the formula specifies the frequencies as a vector and the right hand side factors which classify it.

   The right hand side specifies the log-linear model in the usual form.

   If the resulting array is a ‘ragged’ array the frequencies are padded with structural zeros to become a regular array.

   In this case the formula cannnot be in number form.

Unusual features.

- The fitting function will call an underlying generic function `loglm1`

  Most fitting functions do not use generic functions, but `lme` and `nlme` also `lda` and `qda` from the MASS library are exceptions.

- Well established custom dictates the formula must come first, so method dispatch will have to be on the *second* argument rather than the principal argument.

  This is less unusual with the new style method dispatch, which may nominate a *signature* leading to method dispatch.

Methods for `crosstabs`, `data.frame` and `array` will be useful.

The `data.frame` has to do two jobs:

1. The frequencies have to be placed in the cells of a regular array.

2. If there are more than one frequency for some cells of the array, as defined by the factors, an extra dummy dimension has to be added to the array to accommodate the multiple frequencies.

   In this case the starting value for the iterative process has to be adjusted so that any empty cell of the enlarged table is correctly handled as a structural zero.

Methods for generic interrogator functions

Having an object of class `loglm` allows us to write methods for their convenient handling. The one provided are `anova`, `coef`, `fitted`, `print` (and `show`), `residuals`, `summary` and `update`.

In addition `summary` and `anova` methods for `loglm` objects generate new kinds of objects with their own `print` methods.

A test drive

```
> fm = loglm(~ 1 + 2 + 3, barley.exposed)
> fm1 = update(fm, .~.^2)
> anova(fm, fm1)
LR tests for hierarchical log-linear models

Model 1:
  ~  1 + 2 + 3
Model 2:
 .  ~  1 + 2 + 3 + 1:2 + 1:3 + 2:3

          Deviance  df Delta(Dev) Delta(df)
  Model 1  195.608 106
  Model 2   93.959  56    101.649        50
Saturated    0.000   0     93.959        56

          P(> Delta(Dev)
  Model 1
  Model 2        0.00002
Saturated        0.00112

> getSlots(barley.exposed)
    .Data      .Dim      .Dimnames
 "vector" "integer" "list or NULL"
> r = barley.exposed
> r@.Data = as(resid(fm1), "vector")
Re-fitting to get frequencies and fitted values
> round(r, 1)
```

, , 1

	5/24-6/7	6/7-6/19	6/19-7/2	7/2-7/16	7/16-7/31
alpine	0.48	−0.93	0.14	0.66	−0.24
dicktoo	−1.50	−1.01	1.49	−2.08	1.89
ellis	−0.33	1.00	−1.37	−0.06	0.46
kearney	0.09	0.20	−0.63	−1.24	0.81
minter	0.83	0.14	−0.59	1.16	−3.01
norstar	0.27	0.34	1.22	−1.76	−1.99
reno	−0.58	0.28	−0.41	0.85	−0.12
tenn. wt.	−0.09	−0.02	0.27	−0.30	0.06

, , 2

	5/24-6/7	6/7-6/19	6/19-7/2	7/2-7/16	7/16-7/31
alpine	−0.69	0.29	−0.98	0.01	1.45
dicktoo	1.34	−0.29	−0.44	0.38	−1.92
ellis	0.83	−0.44	0.38	0.04	−2.41
kearney	−0.69	−0.41	0.75	0.61	−0.17
minter	−0.50	0.52	0.01	−0.22	0.34
norstar	0.76	−0.67	0.46	0.20	−2.31
reno	−0.23	0.57	0.20	−0.65	0.07
tenn. wt.	−0.27	0.10	−0.49	−0.45	1.30

, , 3

	5/24-6/7	6/7-6/19	6/19-7/2	7/2-7/16	7/16-7/31
alpine	0.16	0.66	0.77	−0.55	−0.90
dicktoo	−0.08	1.23	−1.31	0.73	−1.28
ellis	−0.55	−0.78	0.68	0.01	0.62
kearney	0.67	0.22	−0.36	0.08	−0.61
minter	−0.47	−0.62	0.42	−0.68	1.38
norstar	−1.28	0.35	−1.24	0.38	1.67
reno	0.86	−0.92	0.17	−0.17	0.06
tenn. wt.	0.39	−0.07	0.23	0.62	−1.04