Guidelines for S3 Regression Models

Stephen Milborrow

June 15, 2015

Abstract

This document presents some guidelines for S3 regression models. Models that follow these guidelines will be compatible with tools that further process the model, such as `plotmo` [3].

Contents

1 Introduction

Once a regression model has been built we usually want to use it for further processing. For example, we may want make predictions from the model. Or we may want to plot the model's response as the predictors are varied (which is what `plotmo` does), or plot the model's residuals (which is what `plotres` does[1]).

For an S3 model to be amenable to such further processing it should follow some commonly accepted interface standards. These are obvious to experienced developers, but there are many packages on CRAN that don't follow them. This is possibly because there seems to be no summary of the standards.

What are the standards? The current document attempts to give a convenient summary, by way of a checklist and an example. More generally, the S programming book by Venables and Ripley [5] still seems to be the best place for advice on writing S3 model code, although a little dated. (For R programming in general there are of course more modern books, but here we are talking specificially about S3 regression models.) The ultimate reference is the R core code itself, and examples in Venables and Ripley should be checked against that code for current practice.

[1]Both those function are in the `plotmo` package [3], but this document is not really about those functions. They are just examples.

2 Checklist for S3 regression models

S3 regression models should adhere to the following guidelines. Some of these may be disregarded in certain situations. This is not a comprehensive list, but enough for most applications.

1. Give the model a unique class: `class(model)` shouldn't return `"list"`. So in the model-building function, do something like `class(model) <- "modelclass"`. In general, the class name should be the same as the model-building function.

2. Save the `call` with the model. In the model-building function, do something like `model$call <- match.call()`.

3. Allow the user to abbreviate argument names and values. Use `match.arg` or similar to match arguments that take strings.

4. Provide both formula and x,y model-building functions. Call the formula method `modelclass.formula` and the x,y method `modelclass.default`.

5. For model functions with a formula interface, save the `terms` with the model. (A `terms` object is a model formula with additional attributes, as described on the help page for `terms.object`. Additional background is given in Chambers and Hastie [1] and Venables and Ripley Section 4.2 [5].)

6. For model functions with an x,y interface:

   i. Use `x` and `y` as the first two arguments to the model-building function. Don't call these arguments anything but `x` and `y`, unless that isn't meaningful for your model.

   ii. The x,y interface should be as similar as possible to the formula interface. Where possible, `summary`, `predict`, and friends should work in the same way for models built with the x,y interface and the formula interface.

      One acceptable difference between the formula and x,y functions is as follows: The formula interface should convert factors in `x` to indicator columns before doing the regression; the x,y interface should reject factors with an error message.[2]

      In the formula interface, conversion of factors comes with the standard use of `model.matrix` (Section 3). In the x,y interface, using `as.matrix` as described below will correctly reject factors and other unsuitable data.

   iii. Be kind to the user and allow `x` and `y` to be `data.frame`s or matrices (not just matrices), even if the model takes only numeric input. Issue a clear error message if `x` or `y` cannot be converted to numeric.

      For the conversion to `matrix` you can use `as.matrix`. This will convert all columns to strings if there are *any* factors or strings in the input. So to check that the input was converted to all numeric as required, it suffices to check just the first element, because either all or none of the converted

[2]The `earth` package [4] treats factors in the x,y interface in the same way as factors in the formula interface (it expands them to indicator columns). For many models that is ideally the way to go, but implementing it is quite lot of work.

matrix elements will be numeric. Note that `as.matrix` is efficient in that it will simply return `x` if `x` is already numeric (it doesn't make a copy of `x`).

Alternatively you can use `data.matrix`. This will convert factors to their internal numeric representation. However, for most models geared towards continuous data it's better to issue an error message than to silently make such conversions, i.e., use `as.matrix` rather than `data.matrix` unless you have a reason not to.

iv. Consider saving `x` and `y` with the model. If you do, save them in fields named `x` and `y`. Don't use those names for anything else saved with the model. If `subset` is supported and specified, save `x` and `y` before taking the subset, and also save `subset`. Likewise for `weights`.

A word of explanation. If the data or environment isn't saved with the model, functions like `plotmo` can't unambiguously access the data used to build the model. Although it works in common scenarios, saving just the `call` isn't sufficient, because one `x` may not be the same as another `x`. (Note that we are talking here about models with an x,y interface. For formula-based models, the `call` and `terms` fields suffice.)

If you are concerned about memory use, give the user an option such as `keep=TRUE` to save `x` and `y`. (There isn't a standard name for this argument — different functions uses different names. In my opinion, please don't follow the precedent set by `lm` and name the argument `x` or `y`; that is inviting confusion.[3])

Note that saving `x` and `y` doesn't use as much memory as one might expect, because R will merely create references to `x` and `y`, not make copies of them. On the other hand, R's automatic garbage collection won't be able to release the memory used by `x` and `y` until the model is deleted.

7. Provide a predict method for the model. The first two arguments for the predict method should be `object` and `newdata`.

   The default `newdata` should be `NULL` and this should be treated as if the user specified the data used to build the model. If that isn't possible unless `keep` (or similar) was set when building the model, issue an error message to that effect.

   The third argument for the predict method should be `type`, unless that isn't meaningful for your model. Make `"response"` one of the options for `type`, possibly the default, unless that isn't meaningful for the model.

   Provide defaults for the other arguments where possible so the user can call `predict` with minimum bother. Be kind to the user and allow `newdata` to be a matrix or a `data.frame`. (From `plotmo`'s perspective this is more than just being kind, it's necessary for `plotmo`'s default predict method.) You can use `as.matrix` for the conversion to `matrix`, as described above.

8. If the model supports prediction or confidence levels, allow the user to access those in the same way as `predict.lm`, i.e., when the appropriate arguments are specified, `predict` should return a matrix with column names `fit`, `lwr`, and `upr`.

[3]This is an exception to the rule that models should conform to the `lm` way of doing things. Note also that `lm.fit` shouldn't be used as an example of an x,y interface — because, for example, `predict` can't be used to make predictions on `lm.fit` models. Instead use a "`.default`" function as illustrated in Section 3.

9. It is good practice to provide the standard model functions. A basic list is `case.names`, `coef`, `coefficients`, `fitted`, `fitted.values`, `model.matrix`, `na.action`, `plot`, `print`, `print.summary`, `resid`, `residuals`, `summary`, `update`, `variable.names`, and `weights`. Not all of those may apply to your model. Many of them come for free if the model is built in the standard way (the default methods in the `stats` package will automatically work for the model).

3 Example S3 Model

Friedrich Leisch's tutorial [2] is an excellent introduction to building R packages. However the minimal `linmod` code in the tutorial, although ideal for the purposes of the tutorial, has limitations that can create issues with functions that further process the model. For example

```
    fit1 <- linmod(Volume~., data = trees)
    predict(fit1, newdata = data.frame(Girth = 10, Height = 80))
```

gives

```
    Error in eval(expr, envir, enclos) : object 'Volume' not found
```

and

```
    fit2 <- linmod(cbind(Intercept = 1, trees[,1:2]), trees[,3])
    predict(fit2, newdata = trees[,1:2])
```

gives

```
    Error in x %*% coef(object) : requires numeric/complex matrix/vector arguments
```

Plotmo methods can be written to work around these issues, but a more general solution is to modify `linmod` as follows.

```
## new version of linmod

linmod <- function(...) UseMethod("linmod")

linmod.fit <- function(x, y) # internal function, not for the casual user
{                            # first column of x is the intercept (all 1s)

    qx <- qr(x)                                # QR-decomposition of x
    coef <- solve.qr(qx, y)                    # compute (x'x)^(-1) x'y
    df.residual <- nrow(x) - ncol(x)           # degrees of freedom
    sigma2 <- sum((y - x %*% coef)^2) / df.residual  # variance of residuals
    vcov <- sigma2 * chol2inv(qx$qr)           # covar mat is sigma^2 * (x'x)^(-1)
    colnames(vcov) <- rownames(vcov) <- colnames(x)
    fitted.values <- qr.fitted(qx, y)

    fit <- list(coefficients  = coef,
                residuals     = y - fitted.values,
                fitted.values = fitted.values,
                vcov          = vcov,
                 sigma         = sqrt(sigma2),
                 df.residual   = df.residual)

    class(fit) <- "linmod"
    fit
}
linmod.default <- function(x, y, ...)
{
    x <- cbind("(Intercept)"=1, as.matrix(x))
    fit <- linmod.fit(x, as.matrix(y))
    fit$call <- match.call()
    fit
}
linmod.formula <- function(formula, data=parent.frame(), ...)
{
    mf <- model.frame(formula=formula, data=data)
    terms <- attr(mf, "terms")
    x <- model.matrix(terms, mf)
    y <- model.response(mf)
    fit <- linmod.fit(x, y)
    fit$terms <- terms
    fit$call <- match.call()
    fit
}
predict.linmod <- function(object, newdata=NULL, ...)
{
    if(is.null(newdata))
        y <- fitted(object)
    else {
        if(is.null(object$terms))               # x,y interface
            x <- cbind(1, as.matrix(newdata))  # columns must be in same order as orig x
        else {                                  # formula interface
            terms <- delete.response(object$terms)
            x <- model.matrix(terms, model.frame(terms, as.data.frame(newdata)))
        }
        y <- as.vector(x %*% coef(object))
    }
    y
}
```

We can try the new code with a few examples:

```
    library(plotmo)
    data(trees)

    fit1 <- linmod(Volume~., data=trees)        # formula interface
    plotres(fit1)

    fit2 <- linmod(trees[,1:2], trees[,3])      # x,y interface
    plotres(fit2)
```

Functions like `print.linmod` in the Leisch tutorial don't need to be modified for `plotmo`, and don't appear in the above code listing.

The new `linmod.formula` function saves the model `terms`, not just the `formula`. Attaching the `terms` to the model is standard practice for formula models.

The `predict.linmod` function now accepts a `data.frame` or a `matrix`. This is what users would expect, and is necessary for `plotmo`'s default predict method.

Note also that `linmod.default` doesn't require the user to manually add an intercept column. There are also a few minor changes to the model fields for closer compatibility with `lm`.

4 Limitations of the example S3 model

Production code should include sanity tests that aren't included in our simple example. For example, to prevent confusing downstream error messages, `linmod.fit` should be extended to check that `x` and `y` are numeric, and contain no NAs. From the user's perspective an error message like

```
Error in linmod.fit(x, y) : NA in x
```

is better than the error message issued by the current code

```
Error in qr.default(x) : NA/NaN/Inf in foreign function call (arg 1)
```

And a message like

```
Error in linmod.default(x, y) : non-numeric column in x
```

is better than

```
Error in qr.default(x) : NAs introduced by coercion
```

Similar tests should be made in `predict.linmod`, which should also check that the new data has the correct number of columns.

Production code would also handle collinearity properly, ensure x and y have conformable dimensions, and take care of details like propagating rownames in the input data to the `residuals` and other returned fields. All this can be done in `linmod.fit`.

References

[1] J. M. Chambers and T. J. Hastie. *Statistical Models in S*. Chapman and Hall/CRC, 1991. Cited on page 2.

[2] Friedrich Leisch. *Creating R Packages: A Tutorial*. Compstat 2008-Proceedings in Computational Statistics, 2008. `http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf`. Cited on page 4.

[3] Stephen Milborrow. *plotmo: Plot a Model's Response and Residuals*, 2015. R package. Cited on page 1.

[4] S. Milborrow. Derived from mda:mars by T. Hastie and R. Tibshirani. *earth: Multivariate Adaptive Regression Splines*, 2011. R package. Cited on page 2.

[5] W. N. Venables and B. D. Ripley. *S Programming*. Springer, 2004. Cited on pages 1 and 2.