Guidelines for S3 Regression Models

Stephen Milborrow

July 2, 2018

Abstract

This document is intended for authors of R functions that build S3 regression models. It describes how these functions should interface to the rest of the world. The intention is to summarize good practice, not to present new techniques.

Models that follow the guidelines summarized in this document will be compatible with tools that further process the model, such as functions for analyzing model residuals or plotting regression surfaces.

Sample linear model code that can be used as a template for new models is supplied.

1 Introduction

Interface consistency is important for building and comparing R regression and classification models across different packages. For example, we should be able to use the same data across different packages without being forced to make arbitrary conversions to the data using `as.matrix` or `as.data.frame`.

Interface consistency is also important if we want to use the model in further processing. For example, we may want to make predictions from different models in a uniform manner, or plot the regression surfaces of different models using a function like `plotmo`[1].

For an S3 model to be amenable to such processing it should follow some canonical and commonly accepted interface standards (e.g. [1, 4]). These may be obvious to experienced developers, but there are many packages on CRAN and GitHub that don't adhere to them.

What are the standards? This document attempts to give a convenient summary, by way of a checklist and examples. The intention is to make explicit good practice and to highlight some common mistakes.

[1]The `plotmo` function [3] is an example of a tool that needs to work across a wide variety of S3 models. I became aware of the issues discussed in this document while making `plotmo` work with dozens of packages over the last decade.

It is assumed that the reader already has some familiarity with creating packages for regression models (e.g. [2]).

2 Checklist for S3 regression models

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

1. Give the model a unique class. In particular, `class(model)` shouldn't return `"list"`. In the model-building function, do something like `class(model) <- "foo"`.

   In general, the class name should be the name of the model-building function. This means, for example, that if the model-building function is `foo`, the `summary` and `plot` methods will be `summary.foo` and `plot.foo`, as expected.

2. Save the `call` with the model. In the model-building function, do something like `model$call <- match.call()`. This expands any argument names that the user abbreviated to their full names.

3. Provide both formula and x,y model-building functions. Name the formula method *`modelclass`*`.formula` and the x,y method *`modelclass`*`.default`. Typically both of these call an underlying function *`modelclass`*`.fit`.

4. 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 [4].)

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

   i. Use `x` and `y` as the first two arguments to the model-building function, in that order. 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 building the model; the x,y interface should reject factors with an error message.

       In the formula interface, conversion of factors comes with the standard use of `model.matrix` (Section 3.1). 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, `vectors` (if one-dimensional), or matrices. That is, automatically convert to a `matrix` internally in the model function; don't force the user to pre-convert the data. Issue a clear error message when this conversion can't be made.

We suggest `as.matrix` is used for the conversion to `matrix`. Note that `as.matrix` converts *all* columns to strings if there are *any* factors or strings in the input. So to check that the input could be correctly converted to a numeric matrix, you need check only that the first element is numeric, because either all or none of the converted matrix elements will be numeric. Note that `as.matrix` is efficient in that it will simply return `x` if `x` is already a matrix (it doesn't make a copy of `x`).

Alternatively you can use `data.matrix` to make the conversion. This will convert factors in a `data.frame` to their internal numeric representation. This conversion implicitly assumes that any factors are ordered with equally spaced levels, which isn't true in general. Therefore 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 special reason not to.

6. 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 specified 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. Apply the `type` argument even with the default `newdata=NULL`; if that isn't possible, issue an error message rather than silently returning bad results.

   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`.

7. 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`.

8. Allow the user to save the data used to generate the model with the model (i.e. save the `data`, or `x` and `y` arguments). Save this information in fields named `data` or `x` and `y`. Don't use those names for anything else saved with the model.

   A word of explanation. When using the model in further processing, we often need to access the data used to build the model. To do that, we refer to the data via the `call` or `terms` saved with the model. But if the data used to build the model changes after the model is built, the saved `call` and `terms` will misleadingly refer to the changed data. To avoid problems like that, it's a good idea to save the data with the model.

   If `subset` is supported, the precedent is to save the data *after* taking the subset.

   For models built with the x,y interface, we recommend that the response is saved as a one-column matrix (not as a vector), with the response name as the column name of the matrix. This allows functions that process the model to easily access the response name for use in plot labels etc.

If memory use is a concern (generally it isn't), give the user an option such as `keep=TRUE` to save the data. (There isn't a standard name for this argument—different functions uses different names. In our opinion, please *don't* follow the precedent set by `lm` and name the argument `x` or `y`; that can cause confusion.[2])

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

9. It is good practice to provide the standard model functions. A basic list is `case.names`, `coef`, `fitted`, `model.matrix`, `na.action`, `plot`, `print.summary`, `print`, `residuals`, `summary`, `update`, `variable.names`, and `weights`. Not all of those may apply to your model. Some 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).

   Note that `coefficients`, `fitted.values`, and `resid` methods are unnecessary, since the standard functions for these dispatch to `coef`, `fitted`, and `residuals`. For inference the following should be added where applicable: `deviance`, `df.residual`, `logLik`, `nobs`, and `vcov`.

10. Don't call `missing()` in your code. Accomplish the same thing by making the default value of the argument `NA` or some other special value, and checking for that value internally. (The use of `missing` in a function complicates code that calls the function — it has to include two different calls to the function, one with the argument and one without. This can get out of hand if `missing` is used on more than argument.)

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

[2] 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 described in item 3 in the above list.

3 Example S3 Models

This section presents three successive refinements of code for building linear models. Impatient readers can skip directly to the final model in Section 3.3.

(i) The first model is from Friedrich Leisch's tutorial on creating R packages [2]. That tutorial discusses the use of S3 methods in model-building functions, and describes `model.frame` and related functions. To those ends, the code in the tutorial is intentionally kept bare-bones and lacks some useful facilities.

(ii) The second model extends the first model slightly. Its `predict` method is more complete, and sufficient for functions like `plotmo`. However its handling of illegal input is inadequate, and its error messages are often unhelpful.

(ii) The third model extends the model further. It meets the guidelines in Section 2, and issues (mostly) clear error messages for illegal input. It has been thoroughly tested and can be used as a template for authors developing new models.

3.1 Model 1: A basic linear model

Friedrich Leisch's tutorial [2] is a good introduction to building R packages, and is recommended for a broader context on some of the ideas discussed in this document.

The bare-bones `linmod` code in the tutorial, although a very good starting point and ideal for the purposes of the tutorial, has limitations that can create problems with functions that further process the model.

For example, making predictions from models built with the code isn't quite plain sailing:

```
data(trees)
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 the puzzling message

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

3.2 Model 2: Extending the basic model

Tools that process S3 models can sometimes be modified to work around the issues mentioned above, but a better solution is to extend the basic linear model. Figure 1 shows a way of doing that.

On models built with Figure 1, the examples in Section 3.1 now work correctly. One way of doing further checks on the model is to run `plotmo` — this checks that the data used to build the model can be retrieved and that `predict` works as expected. (The flat regression surfaces of the linear model aren't of much intrinsic interest; we use `plotmo` here to plot the surfaces merely as a check that the model behaves.)

```
library(plotmo)
data(trees)

fit1 <- linmod(Volume~., data=trees)        # formula interface
plotmo(fit1)                                # plotmo works as expected

fit2 <- linmod(trees[,1:2], trees[,3])      # x,y interface
plotmo(fit2)                                # plotmo works as expected
```

The new `linmod.formula` saves the model `terms`, not just the `formula`. The new `predict.linmod` accepts a `data.frame` or a `matrix`, as users often expect. Note also that the new `linmod.default` doesn't require the user to manually add an intercept column. There are a few minor changes to the model fields for closer compatibility with `lm`.

Functions like `print.linmod` in Friedrich Leisch's tutorial haven't been modified for the new model, and don't appear in Figure 1.

3.2.1 Limitations of Model 2

Production code should include sanity tests that aren't included in Figure 1. Error handling can be substantially improved. To prevent confusing downstream error messages, we should check that the input data can be converted to numeric, and contains no NAs.

For example, with NAs in the input data a message like

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

is better than the message issued with the Figure 1 code

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

With factors in the `x` passed to `linmod.default`, a message like

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

is a lot clearer than the message issued with the Figure 1 code

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

```
## A simple linear model (extended from Friedrich Leisch's tutorial).
## Functions like print.linmod in the tutorial don't appear in the code below.

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)

    y <- as.vector(as.matrix(y))        # necessary when y is a data.frame
    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)
    names(fitted.values) <- rownames(x)

    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, ...)
{
    fit <- linmod.fit(cbind("(Intercept)"=1, as.matrix(x)), y)
    fit$call <- match.call()
    fit
}
linmod.formula <- function(formula, data=parent.frame(), ...)
{
    mf <- model.frame(formula=formula, data=data)
    terms <- attr(mf, "terms")
    fit <- linmod.fit(model.matrix(terms, mf), model.response(mf))
    fit$call <- match.call()
    fit$terms <- terms
    fit
}
predict.linmod <- function(object, newdata=NULL, ...)
{
    if(is.null(newdata))
        y <- fitted(object)
    else {
        if(is.null(object$terms))              # x,y interface
            x <- cbind(1, as.matrix(newdata))
        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
}
```

Figure 1: *A simple linear model (called Model 2 in the text).*
This model is discussed in Section 3.2.

Similar tests should be made in `predict.linmod`. Production quality code should also ensure that the predictors and the response have conformable dimensions, and take care of details like propagating rownames in the input data to the `residuals` and other returned fields. The user should be notified about issues with collinearity.

3.3 Model 3: A complete linear model

More complete code for building linear models is in the file www.milbo.org/doc/linmod.R. We suggest that this file is used as a template for new S3 models (rather than the code in Figure 1). It addresses the limitations mentioned above, and adheres to the guidelines in Section 2.

The following list answers some common questions about the code in the file.

- The code is intended as a template for new kinds of model. A linear model is used just as an example.
- To convert the code to a new model:
    i. Replace all occurrences of "`linmod`" with the new model name.
    ii. Modify `do.linmod.fit`, `do.predict.linmod`, and the method functions (`print.linmod` and friends). The code specific to linear models can be easily separated from the general-purpose code (such as the code that checks and converts the input data).
- The code has been comprehensively tested to check that it behaves appropriately with both legal and illegal inputs. See `test.linmod.R` in the `slowtests` directory of the `plotmo` package.
- Illegal inputs cause error messages (silent failures are avoided). These error messages try to be as clear as possible.
- The following aren't supported (and will cause an error message if attempted):
    i. `NA`s in the response or predictors.
    ii. The `subset`, `weights`, and `na.action` arguments.
    iii. Factor responses (although factor predictors are supported, see below).
    iv. Multiple responses (the response must be a vector or have one column).

  These features are intentionally omitted to keep the example simple. Where necessary they can be added by extending the code in a straightforward fashion.
- Factor predictors are supported by `linmod.formula` (factors are expanded to indicator columns using contrasts in the standard way). Factor predictors aren't allowed by `linmod.default` (`x` must be numeric or logical).
- The code handles the model formula in a basic way is suitable for most applications. See [4] for more advanced handling (including `subset`, `weights`, and `na.action`) and [5] for a more modern treatment of formulas with extensions.
- Non-intercept models are allowed by `linmod.formula` but not by `linmod.default`.

- Additional method functions such as `variable.names.linmod` may be found in the file `www.milbo.org/doc/linmod.methods.R`.

- There is some inconsistency between the way the predict `newdata` is handled for models built with the formula interface and models built with the x,y interface. This is regrettable but not a big deal in practice, and full consistency would require a lot more code. Both types of model accept `newdata` as a `matrix` or a `data.frame`, but differ in details as follows.

  For models built with the x,y interface: The `newdata` must have the same number of columns as the `x` used to build the model. Column order matters. Column names are ignored. The `newdata` may also be a vector (with length a multiple of the number of original columns).

  Models built with the formula interface have the same behavior as `predict.lm`: All variable/column names present in the original data must be present in the `newdata`, and the variable classes must match the originals (otherwise an error message will be issued). The `newdata` columns may be in any order (i.e. what matters is the column names not the column order). Extra columns if present will be silently ignored.

A thanks goes out to Achim Zeileis for feedback on an early version of this document.

References

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

[2] Friedrich Leisch. *Creating R Packages: A Tutorial*. Compstat Proceedings in Computational Statistics, 2008. `https://CRAN.R-project.org/doc/contrib/Leisch-CreatingPackages.pdf`. Cited on pages 2 and 5.

[3] S. Milborrow. *plotmo: Plot a Model's Residuals, Response, and Partial Dependence Plots*, 2018. R package, `https://CRAN.R-project.org/package=plotmo`. Cited on page 1.

[4] W.N. Venables and B.D. Ripley. *S Programming*. Springer, 2000. `http://www.stats.ox.ac.uk/pub/MASS3/Sprog`. Cited on pages 1, 2, and 8.

[5] Achim Zeileis and Yves Croissant. *Extended Model Formulas in R: Multiple Parts and Multiple Responses*. Journal of Statistical Software, 2010. `https://cran.r-project.org/web/packages/Formula/vignettes/Formula.pdf`. Cited on page 8.