Extending emmeans

Russell V. Lenth

November 4, 2017

1 Introduction

Suppose you want to use emmeans for some type of model that it doesn't (yet) support. Or, suppose you have developed a new package with a fancy model-fitting function, and you'd like it to work with emmeans. What can you do? Well, there is hope because emmeans is designed to be extended.

The first thing to do is to look at the help page for extending the package:

```
R> help("extending-emmeans", package="emmeans")
```

It gives details about the fact that you need to write two S3 methods, `recover_data` and `emm_basis`, for the class of object that your model-fitting function returns. The `recover_data` method is needed to recreate the dataset so that the reference grid can be identified. The `emm_basis` method then determines the linear functions needed to evaluate each point in the reference grid and to obtain associated information—such as the variance-covariance matrix—needed to do estimation and testing.

This vignette presents an example where suitable methods are developed, and discusses a few issues that arise.

2 Data example

The MASS package contains various functions that do robust or outlier-resistant model fitting. We will cobble together some emmeans support for these. But first, let's create a suitable dataset (a simulated two-factor experiment) for testing.[1]

```
R> fake = expand.grid(rep = 1:5, A = c("a1","a2"), B = c("b1","b2","b3"))
R> fake$y = c(11.46,12.93,11.87,11.01,11.92,17.80,13.41,13.96,14.27,15.82,
              23.14,23.75,-2.09,28.43,23.01,24.11,25.51,24.11,23.95,30.37,
              17.75,18.28,17.82,18.52,16.33,20.58,20.55,20.77,21.21,20.10)
```

The y values were generated using predetermined means and Cauchy-distributed errors. There are some serious outliers in these data.

3 Supporting `rlm`

The MASS package provides an `rlm` function that fits robust-regression models using M estimation. We'll fit a model using the default settings for all tuning parameters:

[1] I unapologetically use `=` as the assignment operator. It is good enough for C and Java, and supported by R.

```
R> library(MASS)
R> fake.rlm = rlm(y ~ A * B, data = fake)
R> library(emmeans)
R> emmeans(fake.rlm, ~B | A)

A = a1:
 B    emmean        SE df asymp.LCL asymp.UCL
 b1 11.83800 0.4774474 NA  10.90222  12.77378
 b2 23.30000 0.4774474 NA  22.36422  24.23578
 b3 17.80078 0.4774474 NA  16.86500  18.73656

A = a2:
 B    emmean        SE df asymp.LCL asymp.UCL
 b1 14.68344 0.4774474 NA  13.74766  15.61922
 b2 24.71164 0.4774474 NA  23.77586  25.64742
 b3 20.64200 0.4774474 NA  19.70622  21.57778

Confidence level used: 0.95
```

The first lesson to learn about extending emmeans is that sometimes, it already works! It works here because rlm objects inherit from lm, which is supported by the emmeans package, and rlm objects aren't enough different to create any problems.

4 Supporting lqs objects

The MASS resistant-regression functions lqs, lmsreg, and ltsreg are another story, however. They create lqs objects that are not extensions of any other class, and have other issues, including not even having a vcov method. So for these, we really do need to write new methods for lqs objects. First, let's fit a model.

```
R> fake.lts = ltsreg(y ~ A * B, data = fake)
```

4.1 The recover_data method

It is usually an easy matter to write a recover_data method. Look at the one for lm objects:

```
R> emmeans:::recover_data.lm

function (object, ...)
{
    fcall = object$call
    recover_data(fcall, delete.response(terms(object)), object$na.action,
        ...)
}
<bytecode: 0x000000001da53770>
<environment: namespace:emmeans>
```

Note that all it does is obtain the call component and call the method for class "call", with additional arguments for its terms component and na.action. It happens that we can access these attributes in exactly the same way as for lm objects; so, . . .

```
R> recover_data.lqs = emmeans:::recover_data.lm
```

Let's test it:

```
R> rec.fake = recover_data(fake.lts)
R> head(rec.fake)
```

```
   A  B
1 a1 b1
2 a1 b1
3 a1 b1
4 a1 b1
5 a1 b1
6 a2 b1
```

Our recovered data excludes the response variable y (owing to the delete.response call), and this is fine.

Special arguments By the way, there are two special arguments data and params that may be handed to recover_data via ref_grid or emmeans or a related function; and you may need to provide for if you don't use the recover_data.call function. The data argument is needed to cover a desperate situation that occurs with certain kinds of models where the underlying data information is not saved with the object—e.g., models that are fitted by iteratively modifying the data. In those cases, the only way to recover the data is to for the user to give it explicitly, and recover_data just adds a few needed attributes to it.

The params argument is needed when the model formula refers to variables besides predictors. For example, a model may include a spline term, and the knots are saved in the user's environment as a vector and referred to in the call to fit the model. In trying to recover the data, we try to construct a data frame containing all the variables present on the right-hand side of the model, but if some of those are scalars or of different lengths than the number of observations, an error occurs. So you need to exclude any names in params when reconstructing the data.

Error handling If you check for any error conditions in recover_data, simply have it return a character string with the desired message, rather than invoking stop. This provides a cleaner exit. The reason is that whenever recover_data throws an error, an informative message suggesting that data or params be provided is displayed. But a character return value is tested for and throws a different error with your string as the message.

4.2 The emm_basis method

The emm_basis method has four required arguments:

```
R> args(emmeans:::emm_basis.lm)
```

```
function (object, trms, xlev, grid, ...)
NULL
```

These are, respectively, the model object, its terms component (at least for the right-hand side of the model), a list of levels of the factors, and the grid of predictor combinations that specify the reference grid.

The function must obtain six things and return them in a named `list`. They are the matrix `X` of linear functions for each point in the reference grid, the regression coefficients `bhat`; the variance-covariance matrix `V`; a matrix `nbasis` for non-estimable functions; a function `dffun(k,dfargs)` for computing degrees of freedom for the linear function `sum(k*bhat)`; and a list `dfargs` of arguments to pass to `dffun`.

To write your own `emm_basis` function, examining some of the existing methods can help; but the best resource is the `predict` method for the object in question, looking carefully to see what it does to predict values for a new set of predictors (e.g., `newdata` in `predict.lm`). Following this advice, let's take a look at it:

```
R> MASS:::predict.lqs

function (object, newdata, na.action = na.pass, ...)
{
    if (missing(newdata))
        return(fitted(object))
    Terms <- delete.response(terms(object))
    m <- model.frame(Terms, newdata, na.action = na.action, xlev = object$xlevels)
    if (!is.null(cl <- attr(Terms, "dataClasses")))
        .checkMFClasses(cl, m)
    X <- model.matrix(Terms, m, contrasts = object$contrasts)
    drop(X %*% object$coefficients)
}
<bytecode: 0x0000000025fd3af8>
<environment: namespace:MASS>
```

Based on this, here is a listing of an `emm_basis` method for `lqs` objects:

```
R> emm_basis.lqs = function(object, trms, xlev, grid, ...) {
       m = model.frame(trms, grid, na.action = na.pass, xlev = xlev)
       X = model.matrix(trms, m, contrasts.arg = object$contrasts)
       bhat = coef(object)
       Xmat = model.matrix(trms, data=object$model)
       V = rev(object$scale)[1]^2 * solve(t(Xmat) %*% Xmat)
       nbasis = matrix(NA)
       dfargs = list(df = nrow(Xmat) - ncol(Xmat))
       dffun = function(k, dfargs) dfargs$df
       list(X=X, bhat=bhat, nbasis=nbasis, V=V, dffun=dffun, dfargs=dfargs)
   }
```

Before explaining it, let's verify that it works:

```
R> emmeans(fake.lts, ~ B | A)

A = a1:
 B    emmean        SE df lower.CL upper.CL
 b1 11.87278 0.2284451 24 11.40129 12.34427
 b2 23.09278 0.2284451 24 22.62129 23.56427
 b3 17.77278 0.2284451 24 17.30129 18.24427
```

```
A = a2:
 B    emmean        SE df lower.CL upper.CL
 b1 13.91278 0.2284451 24 13.44129 14.38427
 b2 24.06278 0.2284451 24 23.59129 24.53427
 b3 20.50278 0.2284451 24 20.03129 20.97427

Confidence level used: 0.95
```

Hooray! Note the results are comparable to those we had for `fake.rlm`, albeit the standard errors are quite a bit smaller. (In fact, the SEs could be misleading; a better method for estimating covariances should probably be implemented, but that is beyond the scope of this vignette.)

4.3 Dissecting `emm_basis.lqs`

Let's go through the listing of this method, by line numbers.

2–3: Construct the linear functions, `X`. This is a pretty standard standard two-step process: First obtain a model frame, `m`, for the grid of predictors, then pass it as data to `model.data` to create the associated design matrix. As promised, this code is essentially identical to what you find in `predict.lqs`.

4: Obtain the coefficients, `bhat`. Most model objects have a `coef` method.

5–6: Obtain the covariance matrix, `V`, of `bhat`. In many models, this can be obtained using the object's `vcov` method. But not in this case. Instead, I cobbled one together using what it would be for ordinary regression: $\hat\sigma^2(\mathbf{X}'\mathbf{X})^{-1}$, where $\mathbf{X}$ is the design matrix for the whole dataset (not the reference grid). Here, $\hat\sigma$ is obtained using the last element of the `scale` element of the object (depending on the method, there are one or two scale estimates). This probably under-estimates the variances and distorts the covariances, because robust estimators have some efficiency loss.

7: Compute the basis for non-estimable functions. This applies only when there is a possibility of rank deficiency in the model, and `lqs` methods cannot handle that. All linear functions are estimable, and we signal that by setting `nbasis` equal to a 1×1 matrix of `NA`. If rank deficiency were possible, the estimability package (which is required by emmeans) provides a `nonest.basis` function that makes this fairly painless—I would have coded:

```
R> nbasis = estimability::nonest.basis(Xmat)
```

On the other hand, if rank-deficient cases are not possible, set `nbasis` equal to `all.estble`, a constant in the estimability package.

There is a subtlety you need to know regarding estimability. Suppose the model is rank-deficient, so that the design matrix $\mathbf{X}$ has p columns but rank $r < p$. In that case, `bhat` should be of length p (not r), and there should be $p - r$ elements equal to `NA`, corresponding to columns of $\mathbf{X}$ that were excluded from the fit. Also, `X` should have all p columns. In other words, do not alter or throw-out columns of `X` or their corresponding elements of `bhat`—even those with `NA` coefficients—as they are essential for assessing estimability. `V` should be $r \times r$, however: the covariance matrix for the non-excluded predictors.

8-9: Obtain dffun and dfargs. This is a little awkward because it is designed to allow support for mixed models, where approximate methods may be used to obtain degrees of freedom. The function dffun is expected to have two arguments: k, the vector of coefficients of bhat, and dfargs, a list containing any additional arguments. In this case (and in many other models), the degrees of freedom are the same regardless of k. We put the required degrees of freedom in dfargs and write dffun so that it simply returns that value.

10: Return these results in a named list.

5 Hook functions

Most linear models supported by emmeans have straightforward structure: Regression coefficients, their covariance matrix, and a set of linear functions that define the reference grid. However, a few are more complex. An example is the "clm" class in the ordinal package, which allows a scale model in addition to the location model. When a scale model is used, the scale parameters are included in the model matrix, regression coefficients, and covariance matrix, and we can't just use the usual matrix operations to obtain estimates and standard errors. To facilitate using custom routines for these tasks, the emm_basis.clm function function provided in emmeans includes, in its misc part, the names (as character constants) of two "hook" functions: misc$estHook has the name of the function to call when computing estimates, standard errors, and degrees of freedom (for the summary method); and misc$vcovHook has the name of the function to call to obtain the covariance matrix of the grid values (used by the vcov method). These functions are called in lieu of the usual built-in routines for these purposes, and return the appropriately sized matrices.

In addition, you may want to apply some form of special post-processing after the reference grid is constructed. To provide for this, give the name of your function to post-process the object in misc$postGridHook. Again, "clm" objects (as well as "polr" in the MASS package) serve as an example. They allow a mode specification that in two cases, calls for post-processing. The "cum.prob" mode uses the regrid function to transform the linear predictor to the cumulative-probability scale. And the "prob" mode performs this, as well as applying the contrasts necessary to difference the cumulative probabilities into the class probabilities.

6 Exported methods

For package developers' convenience, emmeans exports some of its S3 methods for recover_data and/or emm_basis—use methods("recover_data") and methods("emm_basis") to discover which ones. It may be that all you need is to invoke one of those methods and perhaps make some small changes—especially if your model-fitting algorithm makes heavy use of an existing model type supported by emmeans. Contact me if you need emmeans to export some additional methods for your use.

A few additional functions are exported because they may be useful to developers. They are as follows:

.all.vars(expr, retain) Some users of your package may include $ or [[]] operators in their model formulas. If you need to get the variable names, base::all.vars will probably not give you what you need. Here is an example:

```
R> form = ~ data$x + data[[5]]
R> base::all.vars(form)
```

```
[1] "data" "x"
```

```
R> emmeans::.all.vars(form)
```

```
[1] "data$x"    "data[[5]]"
```

The retain argument may be used to specify regular expressions for patterns to retain as parts of variable names.

.diag(x, nrow, ncol) The base diag function has a booby trap whereby, for example, diag(57.6) returns a 57 × 57 identity matrix rather than a 1 × 1 matrix with 57.6 as its only element. But emmeans::.diag(57.6) will return the latter. The function works identically to diag except for the identity-matrix trap.

.aovlist.dffun(k, dfargs) This function is exported because it is needed for computing degrees of freedom for models fitted using aov, but it may be useful for other cases where Satterthwaite degrees-of-freedom calculations are needed. It requires the dfargs slot to contain analogous contents.

.get.offset(terms, grid) If terms is a model formula containing an offset call, this is will compute that offset in the context of grid (a data.frame).

```
R> .get.offset(terms(~ speed + offset(.03*breaks)), head(warpbreaks))
```

```
[1] 0.78 0.90 1.62 0.75 2.10 1.56
```

.my.vcov(object, ...) In a call to ref_grid, emmeans, etc., the user may use vcov. to specify an alternative function or matrix to use as the covariance matrix of the fixed-effects coefficients. This function supports that feature. Calling .my.vcov in place of the vcov method will substitute the user's vcov. when it is present in ....

7 Support for rsm objects

As an example of how an existing package supports emmeans, we show the support offered by the rsm package. Its rsm function returns an "rsm" object which is an extension of the "lm" class. Part of that extension has to do with coded.data structures whereby, as is typical in response-surface analysis, models are fitted to variables that have been linearly transformed (coded) so that ±1 on the coded scale represents the scope of each predictor.

Without any extra support in rsm, emmeans will work just fine with "rsm" objects; but if the data are coded, it becomes awkward to present results in terms of the original predictors on their original, uncoded scale. The emmeans-related methods in rsm provide a mode argument that may be used to specify whether we want to work with coded or uncoded data. The possible values for mode are "asis" (ignore any codings, if present), "coded" (use the coded scale), and "decoded" (use the decoded scale). The first two are actually the same in that no decoding is done; but it seems clearer to provide separate options because they represent two different situations.

7.1 The `recover_data` method

Note that coding is a *predictor* transformation, not a response transformation (we could have that, too, as it's already supported by the `emmeans` infrastructure). So, to handle the `"decode"` mode, we will need to actually decode the predictors used to construct he reference grid. That means we need to make `recover_data` a lot fancier! Here it is:

```
1  R> recover_data.rsm = function(object, data, mode = c("asis", "coded", "decoded"), ...) {
2         mode = match.arg(mode)
3         cod = rsm::codings(object)
4         fcall = object$call
5         if(is.null(data))
6             data = emmeans::recover_data(fcall, delete.response(terms(object)), object$na.action, ...)
7         if (!is.null(cod) && (mode == "decoded")) {
8             pred = cpred = attr(data, "predictors")
9             trms = attr(data, "terms")
10            data = rsm::decode.data(rsm::as.coded.data(data, formulas = cod))
11            for (form in cod) {
12                vn = all.vars(form)
13                if (!is.na(idx <- grep(vn[1], pred))) {
14                    pred[idx] = vn[2]
15                    cpred = setdiff(cpred, vn[1])
16                }
17            }
18            attr(data, "predictors") = pred
19            new.trms = update(trms, reformulate(c("1", cpred)))   # excludes coded variables
20            attr(new.trms, "orig") = trms       # save orig terms as an attribute
21            attr(data, "terms") = new.trms
22        }
23        data
24    }
```

Lines 2–6 ensure that `mode` is legal, retrieves the codings from the object, and obtain the results we would get from `recover_data` had it been an `"lm"` object. If `mode` is not `"decoded"`, *or* if no codings were used, that's all we need. Otherwise, we need to return the decoded data. However, it isn't quite that simple, because the model equation is still defined on the coded scale. Rather than to try to translate the model coefficients and covariance matrix to the decoded scale, we elected to remember what we will need to do later to put things back on the coded scale. In lines 8–9, we retrieve the attributes of the recovered data that provide the predictor names and `terms` object on the coded scale. In line 10, we replace the recovered data with the decoded data.

By the way, the codings comprise a list of formulas with the coded name on the left and the original variable name on the right. It is possible that only some of the predictors are coded (for example, blocking factors will not be). In the `for` loop in lines 11–17, the coded predictor names are replaced with their decoded names. For technical reasons to be discussed later, we also remove these coded predictor names from a copy, `cpred`, of the list of all predictors in the coded model. In line 18, the `"predictors"` attribute of `data` is replaced with the modified version.

Now, there is a nasty technicality. The `ref_grid` function in `emmeans` has a few lines of code after `recover_data` is called that determine if any terms in the model convert covariates to factors or vice versa; and this code uses the model formula. That formula involves variables on the coded scale, and those variables are no longer present in the data, so an error will occur if it tries to access them. Luckily, if we simply take those terms out of the formula, it won't hurt because those coded predictors would not have been converted in that way. So in line 19, we update `trms` with a simpler model with the coded variables excluded (the intercept is explicitly included to ensure there will be a right-hand side even is `cpred` is empty). We save that as the `"terms"` attribute, and the original terms as a new `"orig"` attribute to be retrieved later. The `data` object, modified or not, is returned. If data have been decoded, `ref_grid` will construct its grid using decoded variables.

7.2 The `emm_basis` method

Now comes the `emm_basis` method that will be called after the grid is defined. It is listed below:

```
 1 R> emm_basis.rsm = function(object, trms, xlev, grid,
 2                               mode = c("asis", "coded", "decoded"), ...) {
 3        mode = match.arg(mode)
 4        cod = rsm::codings(object)
 5        if(!is.null(cod) && mode == "decoded") {
 6            grid = rsm::coded.data(grid, formulas = cod)
 7            trms = attr(trms, "orig")   # get back the original terms we saved
 8        }
 9
10        m = model.frame(trms, grid, na.action = na.pass, xlev = xlev)
11        X = model.matrix(trms, m, contrasts.arg = object$contrasts)
12        bhat = as.numeric(object$coefficients)
13        V = emmeans::.my.vcov(object, ...)
14
15        if (sum(is.na(bhat)) > 0)
16            nbasis = estimability::nonest.basis(object$qr)
17        else
18            nbasis = estimability::all.estble
19        dfargs = list(df = object$df.residual)
20        dffun = function(k, dfargs) dfargs$df
21
22        list(X = X, bhat = bhat, nbasis = nbasis, V = V,
23             dffun = dffun, dfargs = dfargs, misc = list())
24    }
```

This is much simpler. All we have to do is determine if decoding was done (line 5); and, if so, convert the grid back to the coded scale (line 6) and recover the original `"terms"` attribute (line 7). The rest is borrowed directly from the `emm_basis.lm` method in emmeans. Note that line 13 uses one of the exported functions we described in the preceding section. Lines 15–18 use functions from the estimability package to handle the possibility that the model is rank-deficient.

7.3 Exporting the methods

To make the methods available to users of the rsm package, the following code appears in the `NAMESPACE` file:

```
R> if (requireNamespace("emmeans", quietly = TRUE)) {
       importFrom("emmeans", "recover_data", "emm_basis")
       importFrom("estimability", "all.estble", "nonest.basis")
       S3method(recover_data, rsm)
       S3method(emm_basis, rsm)
   }
```

This only has an effect if the user has the emmeans package installed (in which case estimability is also installed, as it is required); otherwise the code is skipped. We need to import the prototypes for recover_data and emm_basis, and register our new methods. Also, packages emmeans and estimability are included in the Imports section of the DESCRIPTION file.

Alternatively, we could simply export the functions recover_data.rsm and emm_basis.rsm without any need to import anything or register methods. It's simpler to do, but makes those functions user-visible and thus they require documentation.

7.4 A demonstration

Here's a demonstration of this new support. The standard example for rsm fits a second-order model CR.rs2 to a dataset organized in two blocks and with two coded predictors.

```
R> library("rsm")
R> example("rsm")   ### (output is not shown) ###
```

First, let's look at some results on the coded scale—which are the same as for an ordinary "lm" object.

```
R> emmeans(CR.rs2, ~ x1 * x2, mode = "coded",
           at = list(x1 = c(-1, 0, 1), x2 = c(-2, 2)))

 x1 x2   emmean        SE df lower.CL upper.CL
 -1 -2 74.98637 0.2984365  7 74.28068 75.69206
  0 -2 76.97747 0.2402529  7 76.40936 77.54558
  1 -2 76.35145 0.2984365  7 75.64576 77.05714
 -1  2 76.79722 0.2984365  7 76.09153 77.50291
  0  2 79.28832 0.2402529  7 78.72021 79.85643
  1  2 79.16230 0.2984365  7 78.45661 79.86799

Results are averaged over the levels of: Block
Confidence level used: 0.95
```

Now, the coded variables x1 and x2 are derived from these coding formulas for predictors Time and Temp:

```
R> codings(CR.rs1)

$x1
x1 ~ (Time - 85)/5

$x2
x2 ~ (Temp - 175)/5
```

Thus, for example, a coded value of $x_1 = 1$ corresponds to a time of $85 + 1 \times 5 = 90$. Here are some results working with decoded predictors. Note that the at list must now be given in terms of Time and Temp:

```
R> emmeans(CR.rs2, ~ Time * Temp, mode = "decoded",
           at = list(Time = c(80, 85, 90), Temp = c(165, 185)))
```

```
 Time Temp   emmean        SE df lower.CL upper.CL
   80  165 74.98637 0.2984365  7 74.28068 75.69206
   85  165 76.97747 0.2402529  7 76.40936 77.54558
   90  165 76.35145 0.2984365  7 75.64576 77.05714
   80  185 76.79722 0.2984365  7 76.09153 77.50291
   85  185 79.28832 0.2402529  7 78.72021 79.85643
   90  185 79.16230 0.2984365  7 78.45661 79.86799

Results are averaged over the levels of: Block
Confidence level used: 0.95
```

Since the supplied settings are the same on the decoded scale as were used on the coded scale, the EMMs are identical to those in the previous output.

8 Conclusions

It is relatively simple to write appropriate methods that work with emmeans for model objects it does not support. I hope this vignette is helpful for understanding how. Furthermore, if you are the developer of a package that fits linear models, I encourage you to include `recover_data` and `emm_basis` methods for those classes of objects, so that users have access to emmeans support.