Extending lsmeans

Russell V. Lenth

September 23, 2014

1 Introduction

Suppose you want to use lsmeans 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 lsmeans. What can you do? Well, there is hope because lsmeans is designed to be extended.

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

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

It gives details about the fact that you need to write two S3 methods, `recover.data` and `lsm.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 `lsm.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 lsmeans 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(lsmeans)
R> lsmeans(fake.rlm, ~B | A)

A = a1:
 B    lsmean        SE df asymp.LCL asymp.UCL
 b1 11.83800 0.4774474 NA  10.90211  12.77389
 b2 23.30000 0.4774474 NA  22.36411  24.23589
 b3 17.80078 0.4774474 NA  16.86489  18.73667

A = a2:
 B    lsmean        SE df asymp.LCL asymp.UCL
 b1 14.68344 0.4774474 NA  13.74755  15.61933
 b2 24.71164 0.4774474 NA  23.77574  25.64753
 b3 20.64200 0.4774474 NA  19.70611  21.57789

Confidence level used: 0.95
```

The first lesson to learn about extending lsmeans is that sometimes, it already works! It works here because `rlm` objects inherit from `lm`, which is supported by the lsmeans 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> lsmeans:::recover.data.lm

function (object, ...)
{
    fcall = object$call
    recover.data(fcall, delete.response(terms(object)), object$na.action,
        ...)
}
<environment: namespace:lsmeans>
```

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 = lsmeans:::recover.data.lm
```

The trickier part is testing it, as it isn't clear that there is a required `data` argument.

```
R> rec.fake = recover.data(fake.lts, data = NULL)
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.

By the way, the `data` argument is handed to `recover.data` if it is specified in the `ref.grid` or `lsmeans` call. It is needed to cover a desperate situation that occurs with certain kinds of 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.

4.2 The `lsm.basis` method

The `lsm.basis` method has four required arguments:

```
R> args(lsmeans:::lsm.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 `lsm.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: 0x000000000c1f2220>
<environment: namespace:MASS>
```

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

```
1  R> lsm.basis.lqs = function(object, trms, xlev, grid, ...) {
2         m = model.frame(trms, grid, na.action = na.pass, xlev = xlev)
3         X = model.matrix(trms, m, contrasts.arg = object$contrasts)
4         bhat = coef(object)
5         Xmat = model.matrix(trms, data=object$model)
6         V = rev(object$scale)[1]^2 * solve(t(Xmat) %*% Xmat)
7         nbasis = matrix(NA)
8         dfargs = list(df = nrow(Xmat) - ncol(Xmat))
9         dffun = function(k, dfargs) dfargs$df
10        list(X=X, bhat=bhat, nbasis=nbasis, V=V, dffun=dffun, dfargs=dfargs)
11    }
```

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

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

A = a1:
 B    lsmean        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    lsmean        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.

4.3 Dissecting `lsm.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, lsmeans provides a fairly pain-free way to handle this—I would have coded:

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

There is a subtlety you need to know, though. 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 throw-out columns of `X` or their corresponding elements of `bhat`, as they are essential for assessing estimability. `V` should be $r \times r$, however: the covariance matrix for the non-ecxluded 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.

4.4 The "honest" version

Because of the inadequacies mentioned above for estimating the covariance matrix, then—lacking any better estimate—I think it's probably better to set it and the degrees of freedom to `NA`s. We will still be able to get the LS means and contrasts thereof, but no standard errors or tests. With that in mind, here's a replacement version:

```
R> lsm.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)
```

```
        V = diag(rep(NA, length(bhat)))
        nbasis = matrix(NA)
        dffun = function(k, dfargs) NA
        list(X=X, bhat=bhat, nbasis=nbasis, V=V, dffun=dffun, dfargs=list())
    }
```

And here is a test:

```
R> lsmeans(fake.lts, pairwise ~ B)

$lsmeans
 B    lsmean SE df asymp.LCL asymp.UCL
 b1 12.89278 NA NA        NA        NA
 b2 23.57778 NA NA        NA        NA
 b3 19.13778 NA NA        NA        NA

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

$contrasts
 contrast estimate SE df z.ratio p.value
 b1 - b2   -10.685 NA NA      NA      NA
 b1 - b3    -6.245 NA NA      NA      NA
 b2 - b3     4.440 NA NA      NA      NA

Results are averaged over the levels of: A
P value adjustment: tukey method for a family of 3 means
P values are asymptotic
```

5 Conclusions

It is relatively simple to write appropriate methods that work with lsmeans 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 `lsm.basis` methods for those classes of objects, and to remember to export them in your `NAMESPACE` file as follows:

```
S3method(myobject, recover.data)
S3method(myobject, lsm.basis)
```