Using *glmulti* with any type of statistical model, with the example of mixed models from package *lme4*

Vincent Calcagno

November 3, 2011

1 General approach

glmulti is a generic function that acts of a wrapper to functions that actually fit statistical models to a dataset (such as *lm*, *glm* or *gls*). glmulti works out-of-the-box with several types of function (such as *lm*, *glm* or *coxph*), but it can in principle be used with any such function *myfittingfunction*, as long as

1. The function receives a model specification in the form of a formula;
2. The function fits the model by maximum likelihood, which can be accessed through the standard *LogLik* function;

Even when the two conditions above are verified, complications arise because, unfortunately, different fitting functions have different conventions regarding how characteristics of the fit should be accessed. Indeed, most of them come from different packages with different authors and there is no common standard so far.

Thus, in order to harnass glmulti to some specific fitting function, one should also provide accessors, i.e. functions that allow *glmulti* to access the information it needs, while taking care of the specifics of the fitting function. In other words, these accessors will interface *glmulti* and the fitting function so that the two can dialogue; they should return information in some standardized way (defined in the *glmulti* package), regardless of the specific function used.

To perform model selection only, one only needs the two conditions above to be verified, which is very easy to achieve in general. At most, one has to define a suitable *LogLik* function, and wrap the fitting function so that it is called with the standard syntax. But to go further, two accessors should be provided:

1. For model averaging, i.e. to obtain multimodel (unconditional) parameter estimates, *glmulti* must be able to access the fitted coefficients and related information from a (fitted) model object. This is taken care of by the *getfit* function.

2. For multimodel prediction, there should also exist a predict function that can be applied on a (fitted) model object, and that behaves like *predict.glm*.

These two steps are not difficult: one can essentially copy-paste one existing S4 method of function *getfit* and edit it to provide a suitable method for class *myfittingfunction*. Similarly, one should provide a suitable *predict.myfittingfunction* function (in the classical S3 way).

These diferent steps are now illustrated for one specific type of statistical models, that do not follow exactly the behavior of lm or glm and thus are not supported out-of-the-box: mixed models and the *lme4* package.

2 An example: using *glmulti* with *lme4*

2.1 Writing a wrapper of the *lmer* function

The *lmer* function from package *lme4* takes model specifications as formulas but adds some specifities compared to lm or glm: the random effects are specified with specific symbols in the formula. One may want to do model selection and model averaging with respect to the fixed effects, but it would not be advisable to shuffle the structure of the random effects (see the several threads on testing significance of random effects).

Hence, the different candidate models will vary in their fixed effects but will have a common random part. We will consider the simple case of one random effect on the intercept, so that the random part would read

$$+(1|x)$$

with x the grouping variable for that random effect.

As *glmulti* will work on fixed effects only, we will use a wrapper function for *lmer* that will take in formulas for the fixed effects and append to them the (constant) random part. Let us call it *lmer.glmulti*. It will just be:

```
lmer.glmulti <- function (formula, data, random = "", ...) {
    lmer(paste(deparse(formula), random), data = data, REML=F,
    ...)
    }
```

Note that the function calls *lmer* with the *REML=F* option. This is because to use AIC or related criteria, we need actual likelihoods and not restricted likelihoods.

Now, we can run *glmulti* with mixed models, using *lmer.glmulti* as the fitting function. This is an example with simulated data:

```
y=runif(30,0,10) # mock dependent variable
a=runif(30) # dummy covariate
```

```
b=runif(30) # another dummy covariate
c=runif(30) # an another one
x=as.factor(round(runif(30),1))# dummy grouping factor
glmulti(y~a*b*c,level=2,fitfunc=lmer.glmulti,random="+(1|x)")->bab
weightable(bab)
plot(bab, type="s")
```

2.2 Providing a getfit method for mer objects

Now, to do model averaging, the *coef.glmulti* function must know how to access parameter estimates (for the fixed effects), standard errors, and degrees of freedom from individual lmer objects. Since the syntax to do this is a bit different from that for *glm* objects, the default *getfit* method will fail.

The getfit method should return a table with three columns: first the parameter estimates, then their standard errors, then the associated degrees of freedom. The latter information is only used to build confidence intervals with small-sample adjustments (see the documentation for *coef.glmulti*). We thus add a getfit method appropriate for *lmer* objects, as follows:

```
setMethod('getfit', 'mer', function(object, ...) {
summ=summary(object)@coefs
summ1=summ[,1:2]
if (length(dimnames(summ)[[1]])==1) {
summ1=matrix(summ1, nr=1, dimnames=list(c("(Intercept)"),c("Estimate","Std.
Error")))
}
cbind(summ1, df=rep(10000,length(summ1[,1])))
})
```

The if part is only here to deal with the null model ($y \sim 1|x$), for some automatic simplifications by R must then be overcome. Note that degrees of freedom were set to an arbitrary high value. This is because there are different ways to compute them in mixed models, and we do not want to get into these details here. This choice means that only asymptotic confidence intervals (i.e. standard ones) will be computed by *coef*, whatever the method retained. It will affect confidence intervals only. You can put in any computation of degrees of freedoms that you would know is appropriate.

Now, we can do model averaging by calling *coef* on the *glmulti* object:

```
coef(bab)
```

Note how the model-averaged importances shown in the earlier plot (with option *type="s"*) can be recovered from the "importance" column of *coef*'s output.

2.3 Providing a *predict* function for *mer* objects

Finally, to do model averaged predictions, we must define an appropriate *predict.mer* function. The *predict* function should, when applied to a fitted model object, behave like *predict.glm*, i.e. return:

- A vector of predicted values for the original sample, or for the new sample (if newdata was specified);
- A vector of associated standard errors (if *se.fit* was set to *TRUE*)

Such a function is not provided in the *lme4* package for understandable reasons: prediction from mixed model is not quite a straightforward topic. It is not the role of *glmulti* to make decisions regarding this topic. Rather, either the developers of the fitting function, or the final user, should make those decisions. This is why no function to do prediction for *mer* objects is builtin in *glmulti*.

In general, one can take example on the code for *predict.lm* or *predict.glm* to write a customs *predict* function. For illustration, here is one function one may decide to use. It would be appropriate only for the specific case of one random effect that we use as example:

```
predict.mer=function(objectmer,random=random, newdata, withRandom=F,se.fit=F, ...){
# only the case of lmer with one random effect on the intercept is handled here
if (missing(newdata) || is.null(newdata)) {
DesignMat <- model.matrix(objectmer) }
else {
DesignMat=model.matrix(delete.response(terms(objectmer)),newdata)
}
output=DesignMat %*% fixef(objectmer)
if(withRandom){
# !!!! all levels of random effects must be present in the new data
z=unlist(ranef(objectmer)) # fitted random effects
if (missing(newdata) || is.null(newdata)) {
Zt<- objectmer@Zt
} else {
Zt<-as(as.factor(newdata[,names(ranef(objectmer))]),"sparseMatrix")
# sparse model matrix for random effects
}
output = as.matrix(output + t(Zt) %*% z)
}
```

```
if(se.fit){
pvar <- diag(DesignMat %*% tcrossprod(vcov(objectmer),DesignMat))
if(withRandom){
pvar <- pvar+ VarCorr(objectmer)[[1]]
}
output=list(fit=output,se.fit=sqrt(pvar))
}
return(output)
}
```

Note that we added a *withRand* argument, controlling whether inference should be entirely conditional on the fitted random effect, or whether uncertainty on the random effect should be considered as well.

Now, we can do model averaged predictions from the glmulti object:

```
predict(bab, se.fit=T, withR=T)
# compare with/without uncertainty on the random effect
plot(predict(bab, withR=T)$fit,predict(bab, withR=F)$fit)
# generate a new sample
neolia=data.frame(a=runif(30), b=runif(30),c=sample(c))
# predictions for the new sample
predict(bab, newdata=neolia, withR=T, se.fit=T)
```