Stirling Coding Club: predicting from LMs, GLMs and GLMMs

Jeroen Minderman

02/04/2019

Introduction

So linear models, in any form, are possibly one of the most frequently used tools in statistics used in ecology, conservation biology, evolutionary biology and ethology. As a collection of methods, they are extremely versatile and relatively easy to implement. In general terms, they take the following form:

$y_i = b_1 + b_2X + \epsilon_i$

Although several packages can produce "effect" plots from fitted models (see e.g. sjPlot, effects), it is useful (and surprisingly straightforward) to be able to replicate this manually. For one, not all model objects work with such packages. Also, quite often they rely on a specific method to compute standard errors, and it is nice to be able to have more control over exactly what you are predicting, and why. As a bonus, by learning this, you can gain a better understanding of how (G)LM(M)s actually work. To demonstrate calculations of predictions from (G)LM(M)s, I will use simulated data throughout, so you get some code to generate test data (with known parameters) as a bonus.

Specifically, I aim to do the following:

- Show how to generate simulated data to fit LMs to.
- Show how to easily calculate predictions from (G)LMs, both using predict() and manually
- Simulate more complex "mixed effects type" data, and fit a GLMM to this
- Show a couple of different methods to calculate predictions from GLMMs.

Simulating "fake" data for a LM

Suppose we have a measured wing lengths and body masses of N = 50 birds, and we are interesting in modelling weight as a function of wing length. We can simulate a dataset as follows:

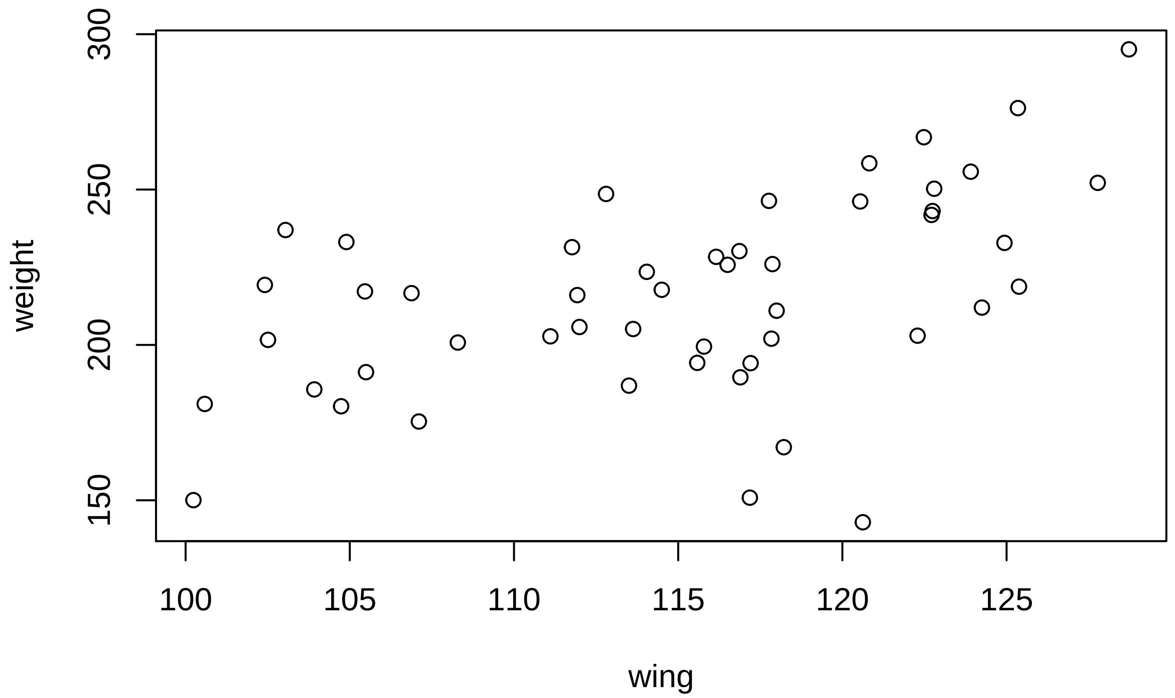
```
wing <- runif(N, 100, 130)
```

So b is the effect on mass of a unit increase in wing length. $N = 50$ is our sample size. $a = 10$ is the predicted weight of a bird with a wing length of zero (clearly nonsense but this is the "intercept" for the model). $e = 30$ is the standard deviation of wing lengths, i.e. the "error" in our model. We then draw a uniform sample of 50 wing lengths between 100 and 130. Once we have this sample of wing lengths, we can draw a sample of weights for each wing length. We assume the mean weight is equal to the intercept plus the slope times b, a basic regression line; the actual weight is drawn from a normal distribution with this mean, and a standard deviation equal to the error e:

```
weight <- rnorm(N, mean = a + b*wing, sd = e)
```

We can now plot this data, fit a linear model to it, and ask for the model summary.

```
plot(weight~wing)
```



```
m <- lm(weight ~ wing)
summary(m)
```

```
##
## Call:
## lm(formula = weight ~ wing)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -84.305 -18.085   3.643  18.978  51.452
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.4193    60.6468  -0.287 0.775176
## wing          2.0283     0.5264   3.853 0.000346 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28.36 on 48 degrees of freedom
## Multiple R-squared:  0.2362, Adjusted R-squared:  0.2203
## F-statistic: 14.85 on 1 and 48 DF,  p-value: 0.0003457
```

Personally, I find that the output from summary() can sometimes be a bit wordy and over-complete (is there such a thing?). The function display() from the package *arm* is a nice alternative, so let's install that.

```
require('arm')
library(arm)
```

```
display(m)
```

```
## lm(formula = weight ~ wing)
##             coef.est coef.se
## (Intercept) -17.42    60.65
## wing          2.03     0.53
## ---
## n = 50, k = 2
## residual sd = 28.36, R-Squared = 0.24
```

So, this model output gives the esntimated intercept, the estimated slope (the estimated effect of wing length on weight), and the associated standard errors. The effect of wing is estimated to be 2.03 and the intercept is estimated to be -17.42. This is pretty close to our parameters b=1.76 and a=10, which means our model is doing a reasonable job. Although obviously this is a trivial example, it is useful to be able to construct more complicated simulated datasets with known parameters, so that you can test model fits on data where you know what the answer should be! We will see more interesting examples later on.

Predicting from LM

Using *predict()*

Although this is obviously a very basic and easy relationship, we can produce predictions from the model we fitted above by using the predict() function. This function has methods for most linear model fits.

```
predict(m)
```

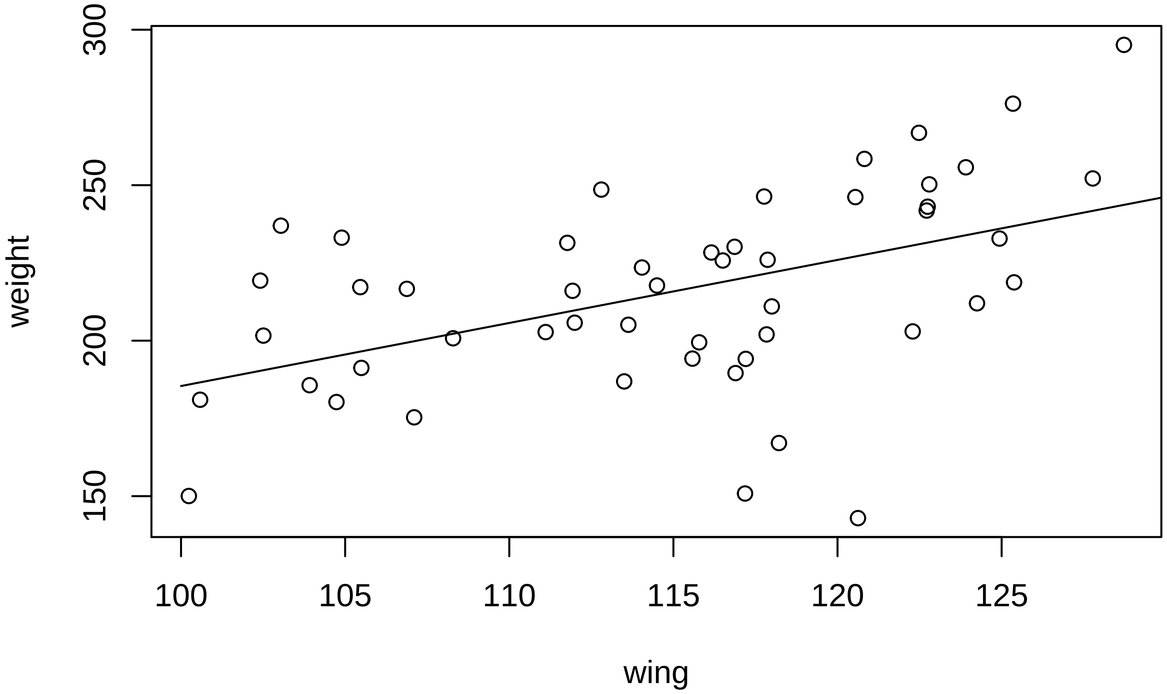
```
##        1        2        3        4        5        6        7        8
## 236.8125 212.7912 236.8811 191.5752 209.5981 207.9369 211.3750 190.4946
##        9       10       11       12       13       14       15       16
## 220.3016 213.8895 209.7298 196.4828 193.3538 199.8134 243.6695 230.6153
##       17       18       19       20       21       22       23       24
## 214.8161 195.3349 233.8993 218.8801 219.6107 209.2736 196.5484 235.9828
##       25       26       27       28       29       30       31       32
## 217.0063 221.4365 220.2567 219.6693 190.3002 231.0026 185.8888 221.9092
##       33       34       35       36       37       38       39       40
## 227.2346 231.5411 222.3526 199.3582 231.6419 217.4219 234.5895 186.5870
##       41       42       43       44       45       46       47       48
## 218.1753 231.4789 221.5881 195.0116 221.6513 227.0732 227.6338 213.0470
##       49       50
## 241.7431 202.2183
```

Using just the predict() function without any further arguments just gives you predictions for the values that were in the observed data; this isn't necessarily very interesting.
Instead, we can produce predicted values for weight given any arbitrary wing length, using the *newdata=* argument. Suppose, for example, we want to predict the wing lengths for birds with wing lengths $wing\_lo$ = 100 to $wing\_hi$ = 130, and store each prediction in a new vector called new_weights:

```
new_wings <- wing_lo:wing_hi
new_weights <- predict(m, newdata=data.frame(wing=new_wings))
```

We now have a series of wing lengths running from 100 to 130 and the corresponding predicting weights for each. Because of this, we can now add a line of these predictions to our earlier plot:

```
plot(weight~wing)
lines(new_wings, new_weights)
```

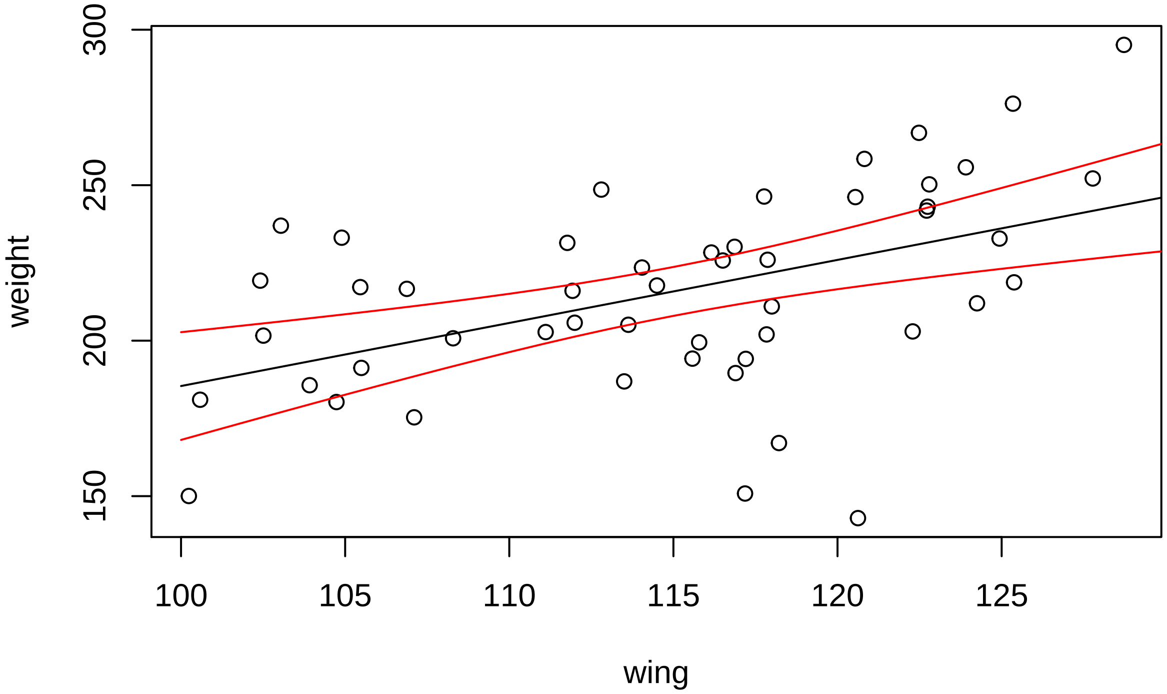


Now, it is possible to extract standard errors of the predictions from the predict() function directly, like so:

```
new_weights <- predict(m, newdata=data.frame(wing=new_wings), se.fit=T)
```

We can now use these SE's by adding and substracting 1.96 times their value and adding a couple of extra lines to the plot, which should then represent +/- upper and lower confidence intervals on our predictions:

```
plot(weight~wing)
lines(new_wings, new_weights$fit) # This is the point predictions
lines(new_wings, new_weights$fit+1.96*new_weights$se.fit, col='red') # Upper CI for prediction
lines(new_wings, new_weights$fit-1.96*new_weights$se.fit, col='red') # Lower CI for prediction
```



Note that because our predictions object *new_weights* now includes multiple elements, we have to refer to the fit and SE elements explicitly using the $ operator.

Calculating predictions 'by hand'

It is useful to be able to replicate these predictions yourself, so that given any model fit, you can calculate these predictions even if *predict()* fails for some reason. To to this, for this example, this is actually really easy. All you need, is the estimated coefficients from the model

```
coef(m)
```

```
## (Intercept)        wing
##  -17.419304    2.028256
```

The predicted value is simply the intercept, plus the slope times the value for wing length you are interested in. We can do this easily by making a matrix with two columns: one a set of '1's to represent the intercept (we always want to multiply with 1) to always get the intercept, and one column with the values for wing length. The number of rows in the matrix will the equal to the number of wing length values we want to predict for. There actually is a built-in function in R to construct a matrix like this, called *model.matrix()*, which takes as an argument the same sort of syntax you would use to fit a linear model. Let's do this for yet another series of wing lengths, a bit of a shorter series this time:

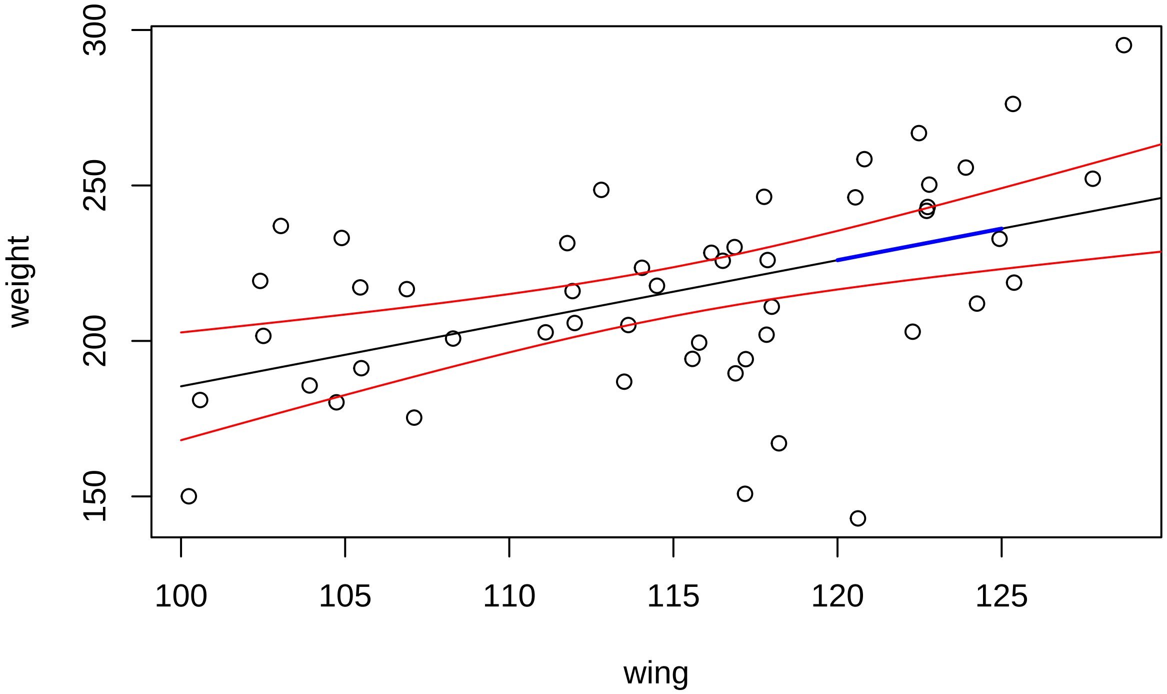
```
more_new_wings <- 120:125
model.matrix(~more_new_wings)
```

```
##   (Intercept) more_new_wings
## 1           1            120
## 2           1            121
## 3           1            122
## 4           1            123
## 5           1            124
## 6           1            125
## attr(,"assign")
## [1] 0 1
```

```
mm <- model.matrix(~more_new_wings)
```

We can now use the matrix multiplication operator to multiply each row of values in this matrix with the coefficients we estimated earlier, to get the prediction we want, and store this as a new set of predictions. When we've done this, we can add this line of predictions to the previous plot to check if we agree with the *predict()* function.

```
more_predictions <- mm %*% coef(m)
plot(weight~wing)
lines(new_wings, new_weights$fit) # This is the point predictions
lines(new_wings, new_weights$fit+1.96*new_weights$se.fit, col='red') # Upper CI for prediction
lines(new_wings, new_weights$fit-1.96*new_weights$se.fit, col='red') # Lower CI for prediction
lines(more_new_wings, more_predictions, lwd=2, col='blue')
```



This of course gives exactly the same result as the values from the predict() function, which is encouraging!

Using *arm::sim()*

Note that obviously with the manual prediction above, we did not also reproduce the standard errors on the predictions as we go from predict() earlier. These are relatively easily calculated analytically using the estimated covariance matrix and the values for which you want to predict (model matrix), but I am not going to go into this in detail here. If you are interested though, the code would be the following:

```
#predict(m, newdata=data.frame(wing=120), se.fit=T)$se.fit  ### Is the same as...
#sqrt(t(mm[1,]) %*% vcov(m) %*% mm[1,])
```

Instead, I'll show how to use the function *sim()* in package *arm* to sample randomly from the estimated parameter distributions, and produce new predictions for each set of sampled parameters. Because this will yield a series of many different predicted lines, we can then derive the average predicted weight for each value of wing, as well as the upper and lower quantiles of this distribution. The result will give us simulated confidence intervals around a prediction.

We begin by drawing 1000 samples from the estimated parameter distributions in model *m*. The resulting object *m_sim* has two elements; $coef for the randomly sampled coefficients, and $sigma which is random samples from the estimated error.

```
m_sim <- sim(m,1000)
head(coef(m_sim)) # This just shows the first 6 rows; one for each simulated draw, and the columns are
```

```
##      (Intercept)     wing
## [1,]    19.88248 1.664543
## [2,]   -84.31995 2.682183
## [3,]    33.68367 1.604071
## [4,]    12.18265 1.720730
## [5,]    49.80146 1.465154
## [6,]   -54.84099 2.389842
```

Now all we need to do, is to repeat the exact same prediction excercise as above (take a matrix to predict with, multiply this with a set of coefficients, and take the sum) *for each line* of the *coef(m_sim)*. This can be done using a loop. Let's first make a new model matrix, with a few more values to predict for

```
more_new_wings2 <- 100:140
mm2 <- model.matrix(~more_new_wings2)
```

We can now use a 'for' loop to take each row of coef(m_sim) (sampled set of parameters), and multiply this prediction matrix with each set. We want to store the resulting vector of predictions as a new row in a dataframe; so we start by initialising an empty data frame. In each loop iteration *i* we then extract row *i* from coef(m_sim), multiply with the model matrix, and use *rbind()* two add the result as a row to the *out* dataframe.

```
out <- as.data.frame(NULL)
for (i in 1:nrow(coef(m_sim)) ) {
  out <- rbind(out, as.vector(mm2 %*% coef(m_sim)[i,]))
}
out[1:5,1:5] # Check out the first few lines and columns
```

```
##   X186.336758531428 X188.001301342942 X189.665844154456 X191.33038696597
## 1          186.3368          188.0013          189.6658         191.3304
## 2          183.8984          186.5805          189.2627         191.9449
## 3          194.0908          195.6948          197.2989         198.9030
## 4          184.2556          185.9763          187.6971         189.4178
## 5          196.3169          197.7820          199.2472         200.7124
##   X192.994929777483
## 1          192.9949
## 2          194.6271
## 3          200.5070
## 4          191.1385
## 5          202.1775
```

```
nrow(out)
```

```
## [1] 1000
```

```
ncol(out)
```

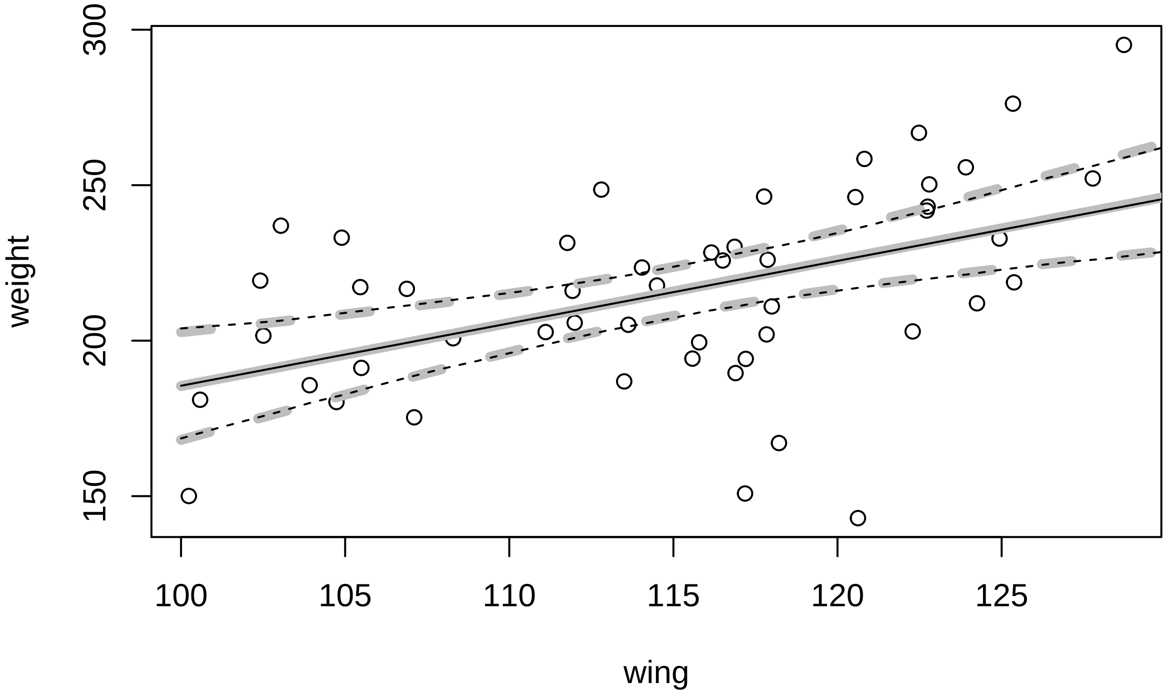
```
## [1] 41
```

Note that to make the above work, we need to make sure to express the prediction result as a vector. The resulting data frame has 1000 rows, one set of predictions for each value of wing in the model matrix. We can now take the mean and upper and lower 95% quantiles of each column, which should give us the equivalent of point predictions and upper and lower confidence bounds of the predictions. We can use the apply function to do this. The third argument of apply is the function to use, the second is to apply it to rows (1) or columns (2), and the first is the data frame or matrix to apply this to.

```
pred_mean <- apply(out, 2, mean)
pred_lower <- apply(out, 2, function(x) quantile(x, probs=c(0.025)))
pred_upper <- apply(out, 2, function(x) quantile(x, probs=c(0.975)))
```

We can now re-plot the point predictions and prediction CI's we made above, and then add the estimated lines from the simulation.

```
plot(weight~wing)
lines(new_wings, new_weights$fit, lwd=5, col='grey') # This is the point predictions
lines(new_wings, new_weights$fit+1.96*new_weights$se.fit, lty='dashed', lwd=5, col='grey') # Upper CI f
lines(new_wings, new_weights$fit-1.96*new_weights$se.fit, lty='dashed', lwd=5, col='grey') # Lower CI f
lines(more_new_wings2, pred_mean)
lines(more_new_wings2, pred_lower, lty='dashed')
lines(more_new_wings2, pred_upper, lty='dashed')
```



Note that the point predictions and upper/lower confidence intervals for the predictions are almost identical to those that we calculated analytically.
This is to be expected in this trivial example, but as models become more complex and data become more noisy, this is not always the case, and the added bonus of the simulation method outlined here, is that it can be applied to any model (or at least many different types of model). In some cases with very complex models, the function to get predict() to return standard errors on the predictions is missing (possibly for good reason!), so it becomes difficult to make prediction plots with confidence bounds by any other means.

Mixed effects models (GLMMs)

Simulating Poisson GLMM data

Mixed effects models are models that use both fixed and random effects. Random effects, instead of obtaining a fixed point estimates for each level in a grouping factor, estimate a single variance, representing variation among groups in the data. The specific "fitted" value for an observation in a given group is then a draw from this distribution.
Fitting mixed effects models can be an effective way of accounting for clustering in data sets, hierarchical data (students within schools, within districts, etc), as well as repeated observations from a single (or more groups). Although they can be fitted relatively easily using a range of packages in R (e.g. nlme, lme4), they can be tricky to understand and tricky to predict from.

To illustrate, I here use the same approach as above, and generate a simulated dataset with observations from multiple groups, with known parameters. These will be as follows:

```
n_groups <- 10      # Number of groups (random effect levels)
n_obs <- 100        # Number of observations per group
group_sd <- 0.5       # Group SD
b <- c(5, 1)      # Fixed effect "true" parameters
```

In the above, the vector b represent the parameters for the “average” linear response in the data (i.e. for a hypothetical “average” group). This is exactly the same as parameters a and b in the example above, but just expressed as a vector. We can now generate a simulated data sets using the same sort of approach as above, but with a few refinements. First, the sample size N is the number of groups times the observations per group. We then draw a random deviation for each group - this represents how different each group is from the overall mean, and is essentially what our random effect is. We then generate a series of X values (the single predictor).

```
N <- n_groups*n_obs
RE <- rnorm(n_groups, mean=0, sd=group_sd)
x <- runif(N, 0.6, 1)
```

Of course we need to add a column with identifiers for group, i.e. ‘group1’, ‘group2’, etc. We can do this by just repeating a number n_obs times and pasting ‘group’ in front of it. We also want to express the result as a factor.

```
groupnames <- paste('group', rep(1:n_groups, each=n_obs), sep='')
groupnames <- factor(groupnames)
levels(groupnames)
```

```
##  [1] "group1"  "group10" "group2"  "group3"  "group4"  "group5"  "group6" 
##  [8] "group7"  "group8"  "group9"
```

To keep things tidy, we can store our simulated data in a data frame.

```
dd <- data.frame(x=x, group=groupnames)
head(dd)
```

```
##           x  group
## 1 0.9791196 group1
## 2 0.8613025 group1
## 3 0.6145506 group1
## 4 0.7535125 group1
## 5 0.9784809 group1
## 6 0.9236544 group1
```

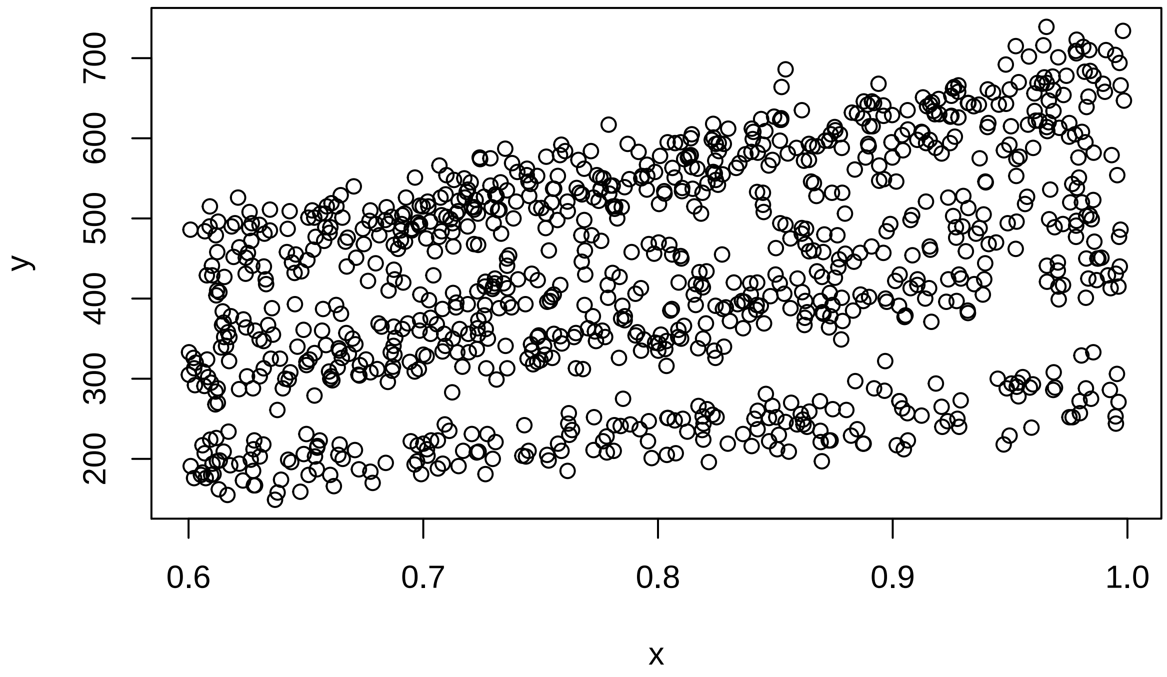
Now it’s time to generate our simulated response variable. To make things more interesting, instead of using a simple normal (Gaussian) response, we here pretend we have observed count data, which come from a Poisson distribution. To make this easier, we can use model.matrix() as above to multiple with our vector of “true” parameters. This gives us *eta0* the response for that “average” group - i.e. the overall mean in the data. We can then add the random effect, i.e. the deviation for each group, to give the specific (mean) observation within each group (*eta*).

```
eta0 <- model.matrix(~x, data = dd) %*% b
eta <- eta0 + RE[dd$group]
```

Because we are dealing with Poisson data, *eta* is the linear predictor, i.e. in this case they are on the log scale. So to get the mean observation within each group, we take the exponential. This mean mu is then used as the mean of a Poisson distribution to draw our observation from. We add these values y to our dataframe and plot y versus x

```
mu <- exp(eta)
dd$y <- rpois(N, lambda = mu)
```

```
plot(y ~ x, data=dd)
```



We now have our simulated dataset, and we can fit a model to it. We here use glmer() from the package lme4. Note the random effect structure specification, and the fact that we specify a Poisson model distribution.

```
glmm1 <- glmer(y ~ x + (1|group), data=dd, family='poisson')
display(glmm1)
```

```
## glmer(formula = y ~ x + (1 | group), data = dd, family = "poisson")
##             coef.est coef.se
## (Intercept) 5.19     0.11
## x           1.01     0.01
##
## Error terms:
##  Groups   Name        Std.Dev.
##  group    (Intercept) 0.33
##  Residual             1.00
## ---
## number of obs: 1000, groups: group, 10
## AIC = 8955.9, DIC = -6914.4
## deviance = 1017.8
```

Our value of the slope for x, 1.01 is almost exactly our “known” value, b[2].

Predicting from GLMMs

The function predict() works with model glmer model fits, just as outlined above for plain linear models. However, there are a number of caveats and things to bear in mind. First and foremost, the function does not return standard errors on the predictions. There are a number of reasons for this, but suffice to say that this limits our immediate ability to plot upper and lower bounds in prediction plots. Second, it is worth asking *exactly what we are predicting* when calculating predictions from mixed effects (or indeed random effect) models. As should be clear from the data simulation excercise above, at its core, a mixed effects model predicts an *average* response for an average group. Leaving aside issues with uncertainty around this prediction for now, such an average prediction may or may not make sense depending on the context in question. In some cases, it is interesting to produce predictions for a hypothetical "average group", even though this specific group did not exist in your data. In others, it may make more sense to predict for a given observed group, or for each group individually. The predict() function itself cannot make these decisions for you, so you have to be explicit about what it is you are doing (and/or are intending to do).

```r
head(predict(glmm1))
```

```
##        1        2        3        4        5        6
## 6.289297 6.169742 5.919350 6.060362 6.288649 6.233013
```

```r
head(predict(glmm1, se.fit=T))
```

```
## Warning in predict.merMod(glmm1, se.fit = T): unused arguments ignored
```

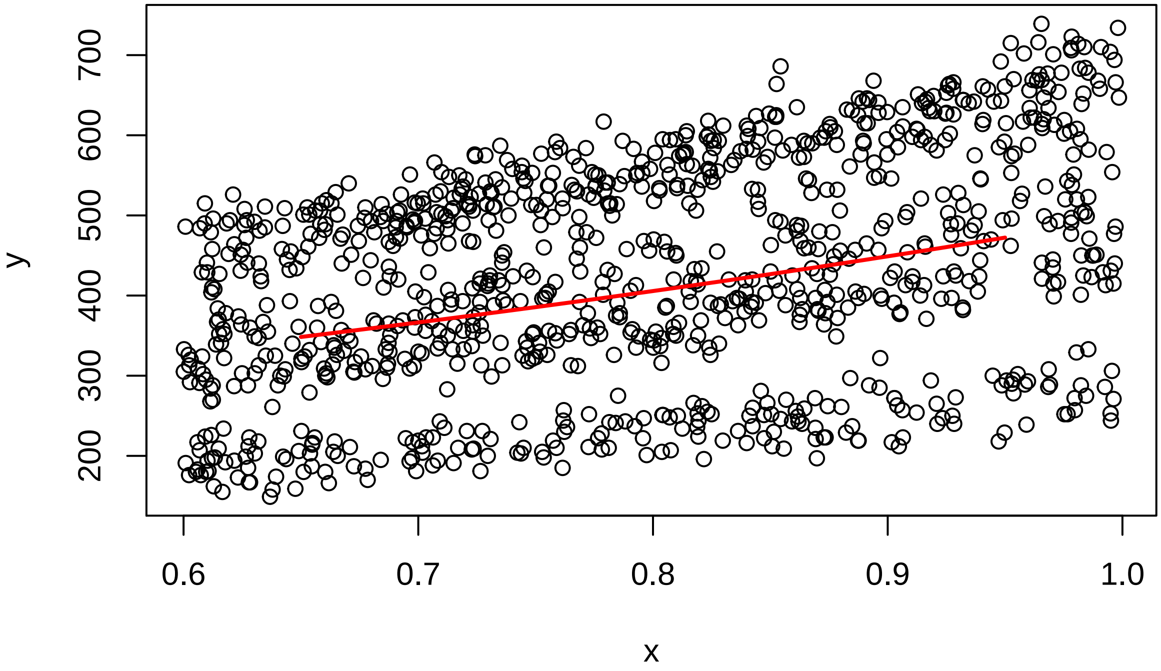
```
##        1        2        3        4        5        6
## 6.289297 6.169742 5.919350 6.060362 6.288649 6.233013
```

Let's make a prediction for a range of x values for an "average" group. Essentially this implies "ignoring" the estimated random effects, so the process is identical to that used for simple linear models. First we set up a range of values of x to predict for, and then we use this in the newdata= argument in the predict() function.

```r
new_x <- seq(0.65,0.95,0.01)
pred1 <- predict(glmm1, newdata=data.frame(x=new_x), re.form=NA, type='response')
```

Note that we have only specified new_x in the new data, and not group. Because of this, to avoid an error message, we have to specify re.form=NA to tell predict to ignore the RE. Also, because we are dealing with a model on a log-link scale, we can use the type='response' argument to back-transform the estimates on the observed scale. We can now plot these predictions as a line in our plot.

```r
plot(y ~ x, data=dd)
lines(new_x, as.vector(pred1), lwd=2, col='red')
```

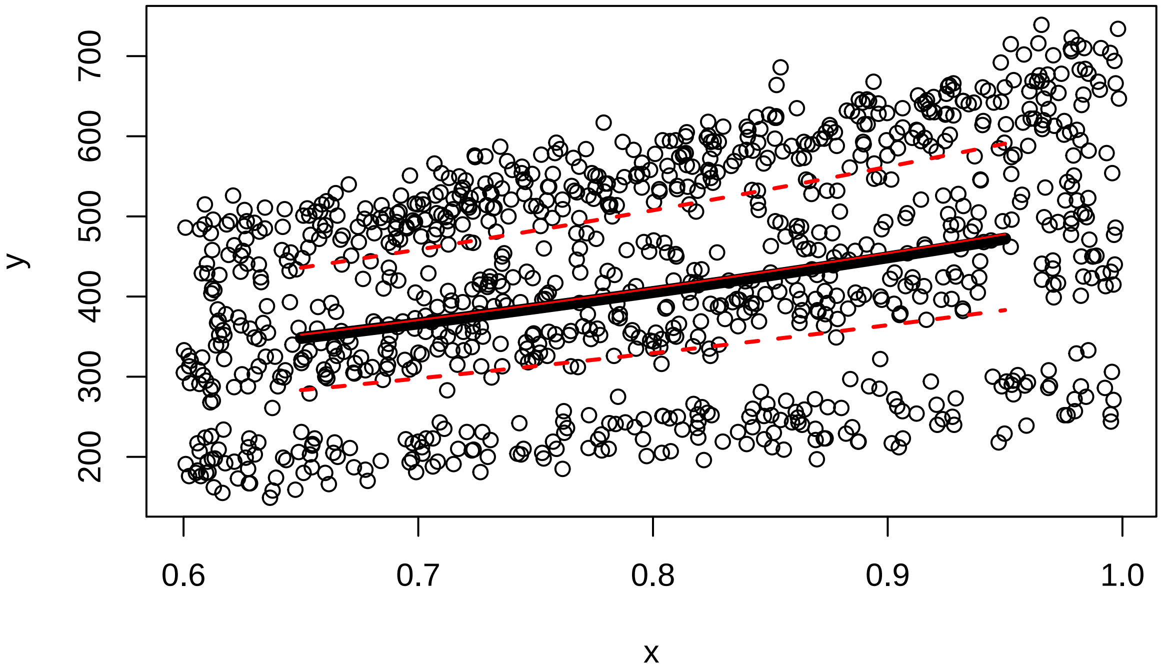


These predictions look sensible as ones for an "average" group. But what about uncertainty around this prediction? We can't use the se.fit values in predict() because they aren't available. As above, we can use *arm::sim()* to obtain a series of samples from the estimated parameter distributions, repeat the predictions many times, and calculate the statistics for each value of X. This is almost the same as the operation before, except we now have to bear in mind that we have samples from both the fixed and random effects to deal with. Also note the back-transformation (exp()).

```
mm3 <- model.matrix(~new_x)     # Make a new model matrix of values new_x
glmm1_sim <- sim(glmm1, 1000)
out <- apply(fixef(glmm1_sim), 1, function(x) as.vector(exp(mm3 %*% x)))
out_mn <- apply(out, 1, mean)
out_lo <- apply(out, 1, function(x) quantile(x, probs=c(0.025)))
out_hi <- apply(out, 1, function(x) quantile(x, probs=c(0.975)))
```

As before, we can now add these predictions to our plot.

```
plot(y ~ x, data=dd)
lines(new_x, as.vector(pred1), lwd=6, col='black')
lines(new_x, out_mn, lwd=1, col='red')
lines(new_x, out_lo, lwd=2, col='red', lty='dashed')
lines(new_x, out_hi, lwd=2, col='red', lty='dashed')
```



Note that the mean of the simulated predictions is exactly the same as the outcome of predict(), as expected. We now alos have upper and lower bounds for each prediction.

It is important to bear in mind with the above, that this method does not take into account the RE or any uncertainty involved with its estimation - in fact we completely ignored it in our predictions just now. Depending on the extent of among-group variation, this may mean that our prediction plot and particularly the estimated uncertainty around the prediction is somewhat optimistic. To take this RE variation into account will take a little bit more work, for example we could use the random draws from the RE's in ranef(glmm1_sim) to generate predictions for all groups, and average across those. However, this is for another day.

Summary

I've given a very brief and very basic overview of how to predict using LMs and GLMMs. This is very far from a comprehensive treatment of the topic, and obviously there are many mathematical details that I have left out of the current text.
It is also important to stress that I have not by any means covered all alternatives to produce uncertainty from model predictions. In particular, bootstrapping is a very useful and universal tool in this regard. There are also numerous packages that are designed to make the entire process described above almost pain-and-code free (e.g. merTools). By all means use them (why reinvent the wheel), but I would suggest that it is always best to have at least a basic grasp of what is likely to go on "under the hood".