Regression basics

Scott Jackson

October 27, 2019

What's this?

This document is part of a course on mixed-effects models (MEM) for the Language Science Center (LSC) at the University of Maryland (UMD). One of the ultimate goals of this course is to equip students with enough knowledge to be able to confidently write code that will generate the kind of hierarchical data for which MEMs are well-suited. The point of this is twofold. On a practical level, being able to simulate complex data that approximates the kind of data one is analyzing in one's actual work is extremely valuable for a lot of applications. On a pedagogical level, being able to write code *from scratch* to generate MEM-appropriate data requires a solid degree of understanding of the structure of MEMs, and it enables further exploration and understanding of the implications and consequences of different analytic scenarios ad assumptions.

As a first step towards this goal, in this document we will go through how to generate data for "simple" (i.e., non-hierarchical, "fixed-effects" only) regression models. In doing so we will work on programming skills as well as a refresher on some basic concepts from regression. But again, the focus is less on mathematical or computational issues and more on practical simulation-based approaches.

The suggested way to interact with this document is as follows:

1. Read through the PDF, and *type along* with the code provided. Going through the effort of typing forces one to pay closer attention to the code than simply running or copy-pasting the given code.
2. Try the Practice Exercises. Only peek at the solutions after you find your own solution or get really really stuck. If your solution differs from mine, that's okay! But try to understand and/or ask why.
3. Attempt the Extention Exercises. It's crucial to try things out with data that you care more about.
4. Submit feedback/questions/gripes/victories as "issues" via GitHub. I may not be able to reply, but I will read them all and try to respond either in class or outside of class.

Good luck!

The modeling approach

The word "modeling" gets thrown around a lot when we talk about data analysis. But what do we mean, and why does it matter? The following is my personal take, so take it with a large heaping helping of salt, but I think it's important to explain where I'm coming from, to help you digest what follows in the rest of this document.

First, especially when we are talking about scientific analysis of data – that is, analysis of data for scientific purposes – our ultimate goal is to construct and test different conceptual models of the world. Our theories and hypotheses imply different models of how the world works, and we want to learn more about which models are better than others.

But it's also important to remember that in a very concrete, immediate sense, all of our models are wrong. Mostly they are wrong because we cannot account for everything, and we are often exploring areas where there are many simultaneous uncertainties. We also often make statistical assumptions that may be reasonable but never exactly satisfied by our real-world data. So we construct models, making many assumptions, and try to advance our understanding by pitting models against each other, on a conceptual/theoretical level, on an empirical level, or both.

In terms of statistical data analysis, many scientists are not trained to think about their analyses as models of their data. Many are – unfortunately in my view – trained to think about statistical analysis as a series of

procedures that will tell them if their experiment "worked" or not. I'm not going to belabor this already opinionated discussion with a full-on rant about why I think this is bad, but in short, I think it leads to a lot of well-meaning but unhelpful approaches to data analysis. So let's just move on by talking about what it means to take a "modeling approach," instead.

What's a model?

What I mean by "model" in the context of data analysis is a mathematical model that can generate data that is comparable to data observed in the world.[1] This covers a lot of ground, but we are going to focus on the family of models relating to the linear model. The point is that in order for a model to be useful for quantitative data analysis, it has to be quantitative. And quantitative models have two major components.

First, we have model structure. This is the part of the model that establishes what the quantities of interest are, at least conceptually, and what the relationships are between those quantities. If you like to think in terms of "boxes and arrows", this is the boxes and arrows part. But it's also the part about the functional form of relationships, like whether a relationship is exponential, quadratic, etc.

Second, we have parameter values. In order for a model to generate data with actual values, there are going to be some kind of parameters that affect what those values are. For example, if your model says that as the height of a person increases, the average weight of the person also increases, there has to be some part of the model that tells you *how much* that increase is.

Some might immediately object, and point out the existence of *nonparametric* models. The deal with nonparametric models, however, is not that they don't *have* parameters (they do), it's that they have varying degrees of flexibility with respect to the assumptions about those parameters, *before the models are fit.* But at the end of the day, if you want a model that will generate data that approximates data one can observe in the world, you have to have some parameters.

Taking this point of view, the "modeling" part of data analysis typically amounts to 1) picking one or more structural models, 2) using software to get parameter values, 3) examining model fit and parameter estimates to build evidence for or against a scientific argument. With this perspective in mind, let's discuss the basics of the linear model, and look at the connection between the data generation process and the results of model fitting.

The linear model

In this course, we are focusing on linear models, eventually building up to versions of these models that have higher "levels" of parameters, which are variously called multi-level models, hierarchical linear models, or mixed-effects models, which for simplicity I will just refer to as MEMs. But especially since we are taking a data simulation approach, it will be helpful to start with the "simpler" case of models without hierarchical parameters.

The linear model itself is a near-ubiquitous tool for data analyis and modeling, in a huge range of applications and fields. For many types of data, the algorithms for estimating these models are straightforward and easy to do, but this class of model has also been generalized to cover a huge range of different situations, in well-established and well-understood ways. In short, the linear model is an extremely flexible, powerful, and robust tool on its own, and understanding it is a good first step to understanding its many more complex extensions.

The basic form of the linear model, and the reason it's called linear, is the following version of a linear equation:

$y_i = \beta_0 + \beta_1 x_{i1} + ... + \epsilon_i$

At the risk of being pedantic, I'll break this down. This says that the ith value of a response variable y is the linear sum of an intercept parameter (β_0), plus one or more products of the corresponding ith value of a predictor variable x_j times the slope parameter for that predictor (β_j), plus an error term that represents all

[1] For language people: thoughout this course I will use the word *data* as a mass noun, not a plural of *datum*.

of the "leftover" (or more technically, "residual") differences between the observed value of y_i and the sum of all of the predictors.

In other words, in order to get the predicted value of any given value of y, you just add up several things. One thing is the *intercept*, which can be thought of as a kind of "default" value, but what it literally means is the predicted value for y when all values of predictors are zero. We'll come back to this, especially when we talk about *centering* variables, but it's essentially the constant in the equation, such that for every observation of y, the intercept is the same, no matter what the values of predictors are. In a scientific context, we often don't really care that much about the intercept, but it's important that it's there. There are models without intercept parameters, but those have slightly different interpretations for the slope parameters, and are much less common. But if you don't have an intercept, you will have to have predictors, and vice versa.

The other thing you *have* to have in your model is the error term, because there will be some difference between your observed response variable and the predicted value, unless all of your data is a perfect function of your intercept plus predictor data (hint: it will never be, unless you have a different predictor for every value of your response, which is a different problem). In other words, while it is hypothetically possible to have zero error, if this happens in real-world data analysis, it almost certainly means something has gone wrong.

Finally, in addition to the minimal structure of intercept plus error, we typically fit models that have one or more predictors, because we would like to examine the relationship between predictor and response. The linear model simply says that as values of each predictor go up or down, the (predicted) values of the response will also go up or down. *How much* the response goes up or down (and which direction it goes) is determined by the β_j parameter, also called the *slope*, for a given predictor.

Simulating and estimating

Switching gears slightly for a bit, let's talk about simulating data. One fundamental way that we can simulate data is to draw random samples from a distribution. This is where software comes in, because software packages like R have built-in ways to draw from a wide range of distributions. For example, there's a simple function that will allow you to generate random samples from the normal distribution, for which you need to specify the mean and standard deviation of the distribution (defaults are 0 and 1, respectively).

```
x <- rnorm(10, mean = 2, sd = 3)
mean(x)
```

```
## [1] 0.7113268
```

```
sd(x)
```

```
## [1] 2.748278
```

```
print(x)
```

```
##  [1]  1.3238054  3.0232215 -1.2182013  0.9380471  1.3209671  3.9724139
##  [7]  4.6391904 -0.7910232 -3.9229560 -2.1721969
```

Technically, because most of us use deterministic computers, "true" random number generation is not possible. R and other software packages use a form of pseudo-random number generation. The goal is essentially to *imitate* random number generation in the confines of a deterministic system. One of the practical benefits of this is that by specifying the "seed" used to generate the values, we can actually replicate processes that depend on "random" samples. Try running this code on your machine, and see if you get the exact same values:

```
set.seed(42)
rnorm(10)
```

```
##  [1]  1.37095845 -0.56469817  0.36312841  0.63286260  0.40426832
##  [6] -0.10612452  1.51152200 -0.09465904  2.01842371 -0.06271410
```

Now let's talk about how to simulate data for a hypothetical linear model.

Generating data from parameters and "recovering" those values

Let's start with the simple case of a continuous response variable (y) and a single, continuous predictor variable (x). In this case, consulting our above equation, we only need three parameters: the intercept (β_0), the slope for x (β_1), and the error. For the error, we actually only need a parameter to represent the standard deviation of the error term (σ).

Let's expand on that last part a bit. The structure of the linear model itself is ambiguous about the nature of the error term. But there is very good theoretical and empirical support (Central Limit Theorem, etc.) for the assumption that errors are normally distributed. Specifically, errors are assumed to have a mean of zero (meaning errors are just as likely to be positive or negative), and the only question is how wide the distribution is, which is quantified by the standard deviation. We will return to this assumption of normally-distributed errors later when we look at evaluating model fit. The point here is that this is a standard assumption, which is why we will generate errors from a random distribution with a single unknown parameter, σ.

So taken together, we can start our data simulation process with just four pieces of information: our three parameters plus the number of observations.

```
beta_0 <- 2
beta_1 <- 3
sigma <- 1
n_obs <- 1000
```

This is the "ground truth" of the data, in the sense that we are playing God and determining the values of the parameters that are the basis of our data. What we want to then establish is that our analysis methods will be able to "recover" these parameter values. That is, if the underlying parameters are known (because we are dictating them), then we can evaluate how well our models do in estimating the correct values of these parameters.

So what we need to do next is build up the right hand side of our linear equation. To do that, we first need to generate a predictor variable x_1. Let's start with another normally-distributed variable, because that's easy, and not a bad assumption (if we are talking about a continuous predictor). In a real data analysis, we often standardize our (continuous) predictors, so we can do that, too. Of course, standardizing doesn't do much when we are already generating x_1 from a normal distribution, but it's not a bad idea to get in the habit of trying to think about what our to-be-analyzed data will look like.

```
x_1 <- as.numeric(scale(rnorm(n_obs)))
```

Now, we can use the linear model to generate y values based on our parameters.

```
y <- beta_0 + beta_1 * x_1 + rnorm(n_obs, mean = 0, sd = sigma)
```

Let's look at a quick scatterplot of `y` by `x_1` to see what that looks like:

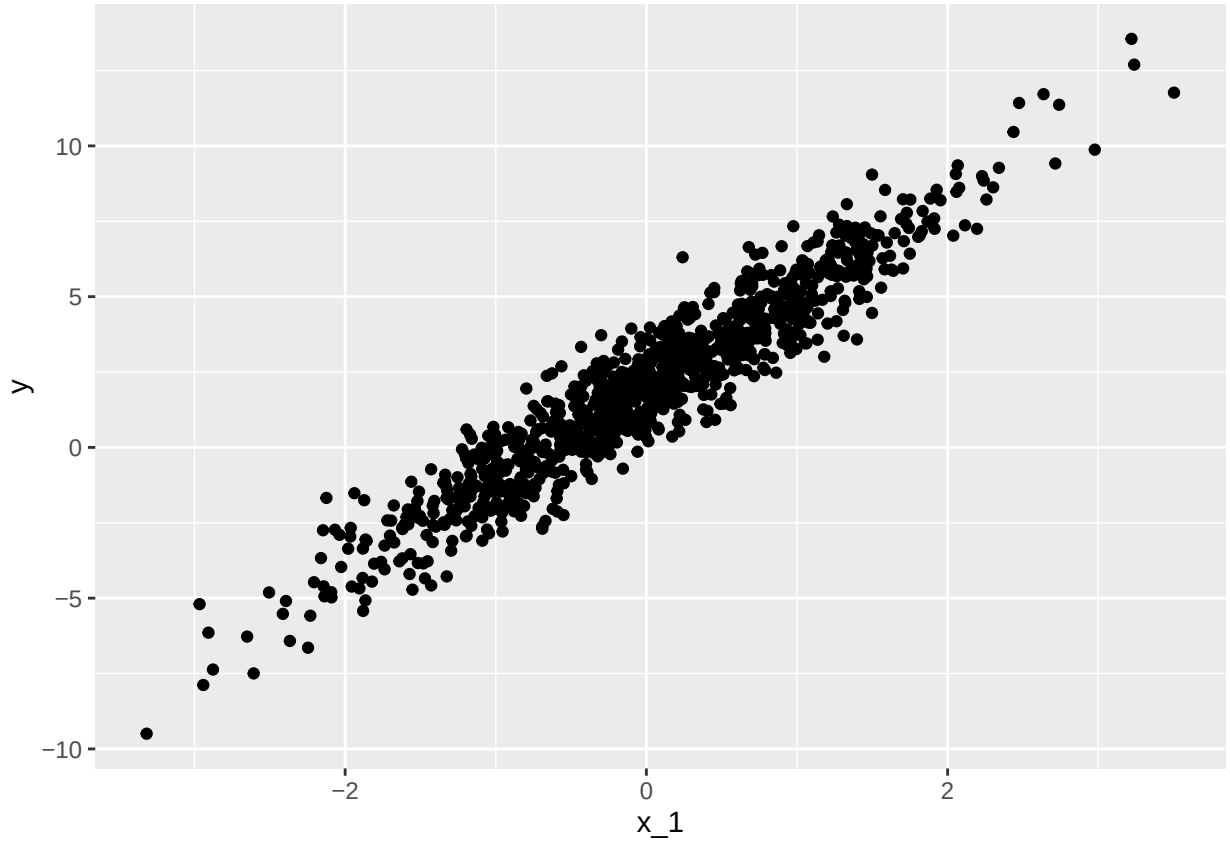
```
library(tidyverse)
```

```
## -- Attaching packages ----------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts -------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
ggplot(data = NULL, aes(x_1, y)) + geom_point()
```



Finally, we can fit a linear model in R, and see whether we got parameter estimates that are close to our ground truth. When we have a fitted model, we usually use the terms *coefficients* for the β parameters and *residuals* for the observation-by-observation errors (mathematically, squared deviations), and this is what they are called in R, as well.

```
fit1 <- lm(y ~ x_1)
fit1$coefficients
```

```
## (Intercept)         x_1
##    1.989090    3.009987
```

```
sd(fit1$residuals)
```

```
## [1] 0.9820724
```

Notice that we get parameter estimates that are very close to the parameters values we set as our ground truth. This is because we have a lot of data, and relatively low error variance for the size of the β parameters we are estimating. Let's explore some different scenarios for practice.

Practice Exercise #1

1. Replicate what we did above, adding two more (continuous) predictors, and using a data frame. That is:
    a. Create variables for all of the parameters needed to generate data from the linear model, including a slope parameter for each of three predictors, as well as the number of observations. Choose different parameter values than those given above.

   b. Generate the predictor data. For now, generate (theoretically) uncorrelated predictors by generating their samples separately (and independently), and putting them in a data frame.
   c. Generate the response data in the same data frame, using the linear model equation.
   d. Fit a model using `lm()`, and pull out the coefficients from the fitted model, to compare with your "ground truth" values.

2. Convert your code from #1 into a function, so that you can pass the parameter values as arguments and get back the estimated parameter values in a list or data frame with appropriate names.

3. Use your new function to explore a few different sets of parameter values. Play around with scenarios where you might not expect the model to get the parameters exactly right, like low N or high error variance.

Extension Exercise #1

Think about how to generate data that is closer to your personal data. For now, ignore any repeated-measures or hierarchical structure. Go through the following questions and think about how you might generate values, or talk about what you don't yet know how to do in order to generate plausible values. By "plausible", I mean that if your response is something like reaction times in milliseconds, then values mostly from -3 to 3 wouldn't make much sense.

1. How would you generate plausible predictor values? Do you have any continuous or (at least somewhat) random predictors, or are all of your predictors determined (like a purely experimental design)? How would you generate experimental predictors/conditions in a data frame?

2. What would some plausible parameter values look like to generate your response variable(s) from your predictors? Are there other things you need to know in order to generate some plausible values?

Repeated simulation

In Practice Exercise #1, you should have played around with simulating some data where you wouldn't expect highly accurate parameter values. Let's do that again here, first defining a function that will let us easily generate data from parameter values. You should have already done this in Practice Exercise #1, but this time we'll just use function with one predictor.[2]

```
sim_simple_lm <- function(n_obs, beta_0, beta_1, sigma) {
  x_1 <- as.numeric(scale(rnorm(n_obs)))
  errors <- rnorm(n_obs, 0, sigma)
  y <- beta_0 + beta_1 * x_1 + errors
  fit <- lm(y ~ x_1)
  output <- list(n_obs = n_obs,
                 beta_0 = beta_0,
                 beta_0_hat = fit$coef["(Intercept)"],
                 beta_1 = beta_1,
                 beta_1_hat = fit$coef["x_1"],
                 sigma = sigma,
                 sigma_hat = sd(fit$resid))
  return(output)
}
```

For illustration, let's pick a really low-power scenario, which does not get close parameter estimates:

```
set.seed(854)
sim_simple_lm(10, 1, 2, 10)
```

[2] In statistical notation, the ^ symbol is often used to represent estimates. So $\hat{\sigma}$ is the estimate of σ and so on. And this symbol often gets pronounced as "hat." So I will sometimes use `hat` in R object names when making this distinction.

```
## $n_obs
## [1] 10
##
## $beta_0
## [1] 1
##
## $beta_0_hat
## (Intercept)
##    7.779565
##
## $beta_1
## [1] 2
##
## $beta_1_hat
##     x_1
## 5.91254
##
## $sigma
## [1] 10
##
## $sigma_hat
## [1] 8.317579
```

So how do we know that we're doing the right thing, when our estimated parameters are so different from ground truth? One answer is to repeat the simulation process. A lot. Because in the limit, the mean parameter value estimates should be very close to the ground truth values. Here's where something like a simple loop structure can be very useful.

Prior to starting the loop, we need to set up some kind of data structure that we can use to "store" the results of each iteration through the loop. In other words, what we are trying to do is to run the data simulation process with the same parameter values over and over again, but each time we run the simulation and analysis, we need to store the parameter estimates, so that when we're done, we can examine the pattern of results across all of the simulations. Luckily this is pretty straightforward.

So the following code sets up an empty `sim_results` data frame, then uses a loop to "fill in" the values. After we have this data frame containing the results from each pass through the loop, we can easily inspect the overall results with something like `summary()`. Compare the means of the simulated parameter estimates to our ground truth parameter values:[3]

```
n_sims <- 1e4 # scientific notatation, for legibility
sim_results <- data.frame(sim = 1:n_sims,
                          beta_0_hat = NA,
                          beta_1_hat = NA,
                          sigma_hat = NA)
for(this_sim in 1:n_sims) {
  if(this_sim %% 1000 == 0) { cat("starting simulation", this_sim, "\n") }
  this_fit <- sim_simple_lm(10, 1, 2, 10)
  sim_results[this_sim, 2:4] <- c(this_fit$beta_0_hat,
                                  this_fit$beta_1_hat,
                                  this_fit$sigma_hat)
}
```

```
## starting simulation 1000
## starting simulation 2000
```

[3]Note that here I am implementing a type of "poor man's progress bar" inside the loop, by having it print out a statement about where it is in the loop every 1000th time through.

```
## starting simulation 3000
## starting simulation 4000
## starting simulation 5000
## starting simulation 6000
## starting simulation 7000
## starting simulation 8000
## starting simulation 9000
## starting simulation 10000
```

```
summary(sim_results[, 2:4])
```

```
##    beta_0_hat       beta_1_hat         sigma_hat
##  Min.   :-9.832   Min.   :-13.2786   Min.   : 2.500
##  1st Qu.:-1.081   1st Qu.: -0.2617   1st Qu.: 7.545
##  Median : 1.006   Median :  1.9765   Median : 9.068
##  Mean   : 1.010   Mean   :  1.9770   Mean   : 9.168
##  3rd Qu.: 3.130   3rd Qu.:  4.2286   3rd Qu.:10.718
##  Max.   :12.642   Max.   : 15.6510   Max.   :17.797
```

Notice that the mean $\hat{\sigma}$ is only close-ish to our ground truth value, while the mean estimates of the β parameters are much closer to their respective ground truth values. This is because σ values themselves are not normally distributed, in part because of the bounded nature of standard deviations (they are always positive). In general, as the N of the data set gets bigger, the distribution of $\hat{\sigma}$ gets closer to normal, so the mean estimated standard deviations will be closer to the ground truth values. The point is that we need to understand our distributional assumptions when we are using simulations to assess whether we're doing the right thing. This is why histograms and visualizations of simulation results are often more useful than summaries like means. Which leads us to our next Exercises.

Practice Exercise #2

1. Practice your data wrangling by plotting histograms of the parameter values in `sim_results`. That is:
   a. Reshape the data so that all of the parameter values are in a single column, with a new column to distinguish the different parameters (`beta_0_hat`, etc.).
   b. Use `ggplot` to plot histograms of the parameters, using `facet_wrap(scales = "free")` to separate the parameters into different sub-plots.
   c. Play around with `binwidth` values.
   d. Play around with colors and/or themes.
2. Take the loop code chunk above and convert the whole thing to a function, also including your plotting code. The function should:
   - take the model parameters (including number of observations) plus the number of simulations as arguments
   - should print out the `summary()` of the simulated parameters as a side effect
   - should generate your plot from #1 as a side effect
   - should return the full data frame of `sim_results` as the return value
3. Use your spiffy new function from #2 and play around with some different parameter values (and number of observations), and look at the impact on the distributions of parameter values. For example, look at how the shape of σ distributions change with the number of observations.

Extension Exercise #2

1. Find or manufacture a linear model analysis in your own data. By "manufacture", I mean you may need to do something you know doesn't make much sense, like a linear regression on a binary response variable. That's fine for our current purposes. Fit this model and save the results as an object so you can examine it. Finally, extract the linear model parameters: intercept, slope(s), residual/error variance, and number of observations.

2. Use the parameters you extracted from #1 to generate data with the same variable names, format, number of observations, etc. as your original data. Turn your code that does this into a function, so that you can easily generate new sets of data with the same or different parameters.
3. Compare your original data to the generated data, using plots, summary statistics, or other methods. How are they similar or different? What might you do to make your generated data more similar? Attempt at least one way of making your generated data look more like your original data.
4. Think about an alternative hypothesis for the model you fit. For example, a slope of zero for a slope parameter that was large in the fitted model, or vice versa.
    a. Generate data under this alternative hypothesis and examine the data.
    b. Use a loop to generate many sets of data under this alternative hypothesis, and save (at least) the modified parameter each time.
    c. Look at how often the parameter of interest matches or exceeds the original value (i.e., by chance).
    d. Do the results make sense? If the alternative parameters never by chance replicate the original values, also try an alternative that is not so different.

For example, if your original fit model had a slope parameter with a value of 3.5 and you are investigating an alternative by generating data with a value of 0 for that slope, generate and fit models 10,000 times and examine how often the fitted parameter reached or exceeded a value of 3.5, even though you generated the data with a "ground truth" value of 0. And if your generated data never by chance results in a slope of 3.5 or greater in 10,000 simulations, try generating data with a ground truth slope of 2.5 or 3.0 instead of 0, and see how often the empirical model fits match or exceed a slope of 3.5.

Categorical predictors

At this point, we have the skeleton of the procedures we will use throughout this course. We established a model in terms of both structure and plausible parameter values, we generated data from that model, we fit a model to the generated data, and we compared the model estimates to our "ground truth," using many simulated data sets and model fits as needed to get a better picture of how well the generation and fitting results are lining up. These are tools we can use to understand, test, and evaluate our model assumptions and empirical model fits.

However, we also started with the simplest case, which almost certainly isn't sufficient for your own research, and isn't even sufficient for this course, since this is a course on MEMs! Throughout the rest of this tutorial, and the rest of the course, we will gradually add to our repertoire so that we can ultimately generate the type of data that matches the models we want to fit in our actual research.

A first step is categorical factors. Especially in experimental research, continuous predictors like we have been using are often the exception rather than the norm, and they are often relegated second-class status as "covariates" rather than factors/predictors of interest. As an aside, in terms of the types of models we are focusing on, the notion of "covariate" has no special status. Those are predictors in our linear model, just like any other. We may decide, for example, that we do not want to include those predictors in interactions with our other predictors or factors, but that is a theoretical decision, and one we could make about any other predictor if we wanted.

So for our purposes, the only difference between continuous and categorical predictors is how we generate the appropriate values. For example, `rnorm()` is not a helpful function for generating categorical values! So let's look at a couple of techniques for generating categorical (predictor) data.

Categories and dummy-coding

Take a moment to think about (nominal) categorical data as a predictor in a linear model. Let's say we have a factor that represents pre- and post-training status, so that we have a factor in our data that is a series of "pre" and "post" labels. Ultimately we would like to fit a model to estimate the "effect" (or "difference") between pre and post. In terms of our linear model, we need to estimate the slope (β) parameter. But what

does the slope multiply in our equation? Maybe this is a little too obvious, but "post" is not a number we can multiply by a slope!

The answer is that categorical factors in linear models are implemented using what is typically called "dummy" coding. That is, we convert our nominal predictors into numeric predictors, often using values of 0 and 1. But how do we choose the values that map to different categories? That depends entirely on how you want to interpret the model parameters.

Let's return to the `sleep` data for illustration. Recall that there is a categorical `group` factor. We start by creating a local copy of the data, and using `levels()` to re-code the factor labels of `group`.

```
mysleep <- sleep
levels(mysleep$group) <- c("pre", "post")
head(mysleep)
```

```
##   extra group ID
## 1   0.7   pre  1
## 2  -1.6   pre  2
## 3  -0.2   pre  3
## 4  -1.2   pre  4
## 5  -0.1   pre  5
## 6   3.4   pre  6
```

```
summary(mysleep)
```

```
##      extra         group          ID
##  Min.   :-1.600   pre :10   1      :2
##  1st Qu.:-0.025   post:10   2      :2
##  Median : 0.950             3      :2
##  Mean   : 1.540             4      :2
##  3rd Qu.: 3.400             5      :2
##  Max.   : 5.500             6      :2
##                             (Other):8
```

Now let's fit a linear model predicting `extra` with the `group` variable.

```
summary(lm(extra ~ group, data = mysleep))
```

```
##
## Call:
## lm(formula = extra ~ group, data = mysleep)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.430 -1.305 -0.580  1.455  3.170
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.7500     0.6004   1.249   0.2276
## grouppost     1.5800     0.8491   1.861   0.0792 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.899 on 18 degrees of freedom
## Multiple R-squared:  0.1613, Adjusted R-squared:  0.1147
## F-statistic: 3.463 on 1 and 18 DF,  p-value: 0.07919
```

So what did R do? What values did it assign to the categorical value? The hint is that the level "post" is

indicated in the summary table next to the coefficient. Think back to the interpretation of linear model coefficients. The interpretation of the intercept is the predicted value of y, *when all predictor values are zero*. The slope parameter β_1 tells you how much the predicted value of y changes *for every unit change in the predictor* x_1. So in this case, if the level "pre" is assigned a value of 0, the intercept is interpreted as the predicted value of `extra` for the "pre" condition. Because there are no other predictors, this corresponds to the mean value of `extra` for the "pre" condition. And if the level "post" is assigned a value of 1, then the slope parameter is interpreted as the *change* in value of y associated with going from "pre" to "post." In other words, this is the estimated difference between the means of "pre" and "post" conditions.

We can confirm this interpretation by comparing the coefficient values with the actual cell means in the data:

```
mysleep %>% group_by(group) %>% summarize(mean = mean(extra))
```

```
## # A tibble: 2 x 2
##   group  mean
##   <fct> <dbl>
## 1 pre    0.75
## 2 post   2.33
```

We can further confirm this by creating a new `group.num` column that is literally 0s and 1s, corresponding to the levels of `group`, and re-fitting the model, getting the same results as before.[4]

```
mysleep <- mysleep %>% mutate(group.num = as.numeric(group) - 1)
summary(lm(extra ~ group.num, mysleep))
```

```
##
## Call:
## lm(formula = extra ~ group.num, data = mysleep)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.430 -1.305 -0.580  1.455  3.170
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.7500     0.6004   1.249   0.2276
## group.num     1.5800     0.8491   1.861   0.0792 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.899 on 18 degrees of freedom
## Multiple R-squared:  0.1613, Adjusted R-squared:  0.1147
## F-statistic: 3.463 on 1 and 18 DF,  p-value: 0.07919
```

This straightforward mapping of interpretations is why 0 and 1 are so commonly used as default values when we include categorical factors in a linear model.

Centering categorical predictors

So if this coding scheme is so straightforward, why would we want to change it? Again, this depends entirely on how we would like to interpret our coefficients. One example is *centering* of variables. Recall from our previous discussion of variable scaling that centering a variable means transforming it such that the mean is zero (or extremely close to zero, such as when we apply R's `scale()` function). Now, think about how this relates to the interpretation of our linear model coefficients. How do we interpret the intercept?

[4] Remember that **factors are numbers with labels**? This means that when we coerce a factor to a numeric vector in R, it gets converted to the integers corresponding to the numbering of the levels (first level is a 1, second level is a 2, etc.). So if you want level 1 to be 0 and level 2 to be 1, you need to subtract 1 from the coerced numeric vector.

The answer is explained by how I defined the intercept above: *the intercept is the predicted value of y when all predictors have a value of zero.* This is just simple arithmetic because when a value of a predictor is zero, then it adds nothing to the predicted value of y because a slope parameter times zero is obviously always zero. So if all predictors are zero, the only terms left in our equation are the intercept and error terms. Now, think about the impact of centering a categorical predictor like `group`. In our previous model, the intercept corresponded to the mean value of the "pre" condition. What does the intercept correspond to when we center the `group` variable? Let's find out in the next Exercise.

Practice Exercise #3

1. Using the `sleep` data, create a new variable corresponding to a centered version of the `group` variable, calling it `group.c`. Hint: how can you transform the `group` variable so that its mean value is 0?
2. Fit a model with the new `group.c` predictor as the only predictor of `extra`. How is it different and similar to the original model with `group` or `group.num`?
3. What does the new intercept correspond to? Calculate that value with another method to demonstrate this interpretation of the coefficient.

Extension Exercise #3

1. Find (or manufacture) a (2-level) categorical factor in your data. If you have only continuous predictors, create a dichotomized version of that variable (hint: remember that `ifelse()` is your friend here). Use `summary()` or another method to examine how the values are distributed in your data (i.e., balanced, or not)?
2. Fit a simple linear model using this factor as your only predictor. Compute cell means of your response variable by the levels of your predictor to confirm the interpretation of the model coefficients.
3. Create a centered numeric version of your factor, and re-fit the model. Compute the corresponding values in your data to match the coefficient values in this new model. If they do not match, can you figure out why not?

Factor coding, continued

So far so good, but what happens if we have a factor with more than two levels? This is where "dummy" coding comes into play. Cutting to the chase, the standard approach is that if you have a factor with N levels, you need $N - 1$ dummy variables to represent that factor. Again, R will perform this coding "behind the scenes" if you use a function like `lm()` with a factor predictor. R's default behavior is called "treatment coding", which simply means that the first level of your factor is treated as the "reference level", and then there is essentially a dummy variable for each other level, where a value of 1 indicates the presence of that level in the data.

We can see an example of this if we re-visit the `iris` data. We can use the function `contrasts()` to see what the default contrast coding is for a the three-level factor of `Species`.

```
contrasts(iris$Species)
```

```
##            versicolor virginica
## setosa              0         0
## versicolor          1         0
## virginica           0         1
```

The way to read this table is that the left-hand side represents the values of the `Species` variable, and the columns represent the dummy-coded variables. Reading across the table tells you how each factor level is coded in the dummy variables. So the `versicolor` and `virginica` levels are both coded for a value of 1 in the respective `versicolor` and `virginica` dummy factors, and because `setosa` is the reference level, it is coded as zero in both.

Think about how this maps to the discussion above about the interpretation of coefficients in the following exercises.

Practice Exercise #4

1. Create a local copy of the `iris` data.
2. Fit a model predicting `Sepal.Width` using the `Species` factor.
3. Create dummy variables to re-create the implicit coding from the model in #2, and re-fit the model using those new variables instead of `Species`. Confirm that the coefficient estimates are identical to those in #2.
4. Use `mutate()` and `relevel()` to create another version of the `Species` factor, where the value of "versicolor" is the reference level.
5. Re-fit the model from #2 using the new factor from #4. Explain the differences between coefficients.
6. Adding only a single new dummy variable, re-fit the same model as #5, using dummy-coded variables instead. Confirm that the coefficient estimates are identical.

Extension Exercise #4

1. Find or manufacture a factor with > 2 levels in your own data. Aim for > 3, if possible, to get practice with more levels.
2. Create dummy-coded variables to represent this factor.
3. Fit a model using both the original factor and the dummy-coded factors, and confirm that they get identical coefficient estimates.
4. Fit a model using fewer of the dummy variables. How can we interpret the coefficients? Can we double-check this somehow?

Simulating data with categorical predictors

Now that we better understand the underlying structure of how factors are coded, and how those values map to the linear model equation, we are in a position to be able to simulate data of this kind.

The biggest trick is figuring out how to create the dummy-coded vectors in a way to match the structure you want in the data. Let's look at a couple of methods in the next exercise.

Practice Exercise #5

1. Let's simulate some data similar in structure to the `iris` data, focusing on the `Sepal.Width` model from earlier. Make another fresh local copy of the data.
2. Fit model with `Species` predicting `Sepal.Width`, and take note of the model parameters, out to two or three decimal places.
3. Create a data frame consisting of a `plantID` column that goes from "plant1" to "plant150", and two columns that represent `versicolor` and `virginica` dummy-coded factors. Try two different methods. With both methods, use a function like `xtabs()` or `group_by()`/`summarize()`
    a. use `rep()` to generate vectors of 0s and 1s
    b. use `contr.treatment()` and take advantage of "recycling"
4. Adapt one of your previous functions to generate data from linear model parameters (specifically, a version with two slope parameters). Using the parameters from the model you fit in #2, generate data.
5. Fit a model to this simulated data and compare to the original fitted model.

6. Compare plots of the simulated and original data (using boxplots). Hint: try to put both into a single data frame, and try to plot the corresponding boxes next to each other.

Extension Exercise #6

Replicate the process for Practice Exercise #6 with your own data. Suggested steps:

1. Fit a model in your data, take note of the parameters needed to generate data from a linear model.
2. Re-use or adapt functions you have already created to generate data from a model with these parameters. This involves essentially two steps:
    a. Generate predictor values, based on how the predictors are structured in your data.
    b. Use the linear model equation to generate response values from these predictors and your parameters.
3. Compare a model fit to this simulated data to confirm that it's similar to your initial model. If needed, use a loop to run many simulations, in case you want to see whether the long-run (fitted) estimates of your parameters matches your "ground truth" parameters.
4. Compare plots of your simulated vs. real data, trying to understand any potential differences.

The End (for now)

We have taken first steps into simulating data given model parameters. From here on out, we will be following similar procedures, but will be adding more complexity and different options, to cover more modeling situations. Next we will examine interaction terms and logistic response models, before we move on to MEMs.