

Statistical Rethinking with brms, ggplot2, and the tidyverse

version 1.0.1

A Solomon Kurz

2019-05-03

Contents

This is a love letter	7
Why this?	7
My assumptions about you	7
How to use and understand this project	8
You can do this, too	8
We have updates	8
1 The Golem of Prague	11
1.1 Statistical golems	11
Reference	12
Session info	12
2 Small Worlds and Large Worlds	13
2.1 The garden of forking data	13
2.2 Building a model	28
2.3 Components of the model	30
2.4 Making the model go	33
Reference	40
Session info	40
3 Sampling the Imaginary	41
3.1 Sampling from a grid-like approximate posterior	42
3.2 Sampling to summarize	43
3.3 Sampling to simulate prediction	55
3.4 Summary Let's practice in brms	64
Reference	66
Session info	66
4 Linear Models	69
4.1 Why normal distributions are normal	69
4.2 A language for describing models	75
4.3 A Gaussian model of height	76
4.4 Adding a predictor	91
4.5 Polynomial regression	105

Reference	110
Session info	110
5 Multivariate Linear Models	113
5.1 Spurious associations	113
5.2 Masked relationship	130
5.3 Multicollinearity	138
5.4 Categorical variables	149
5.5 Ordinary least squares and <code>lm()</code>	157
Reference	157
Session info	157
6 Overfitting, Regularization, and Information Criteria	161
6.1 The problem with parameters	161
6.2 Information theory and model performance	170
6.3 Regularization	180
6.4 Information criteria	183
6.5 Using information criteria	189
6.6 Summary Bonus: R^2 talk	200
Reference	202
Session info	202
7 Interactions	205
7.1 Building an interaction.	205
7.2 Symmetry of the linear interaction.	214
7.3 Continuous interactions	215
7.4 Interactions in design formulas	222
7.5 Summary Bonus: <code>marginal_effects()</code>	222
Reference	228
Session info	228
8 Markov Chain Monte Carlo	231
8.1 Good King Markov and His island kingdom	231
8.2 Markov chain Monte Carlo	233
8.3 Easy HMC: <code>map2stan brm()</code>	233
8.4 Care and feeding of your Markov chain.	243
Reference	251
Session info	251
9 Big Entropy and the Generalized Linear Model	255
9.1 Maximum entropy	255
9.2 Generalized linear models	266
Reference	276
Session info	276

10 Counting and Classification	279
10.1 Binomial regression	279
10.2 Poisson regression	301
10.3 Other count regressions	312
Reference	319
Session info	319
11 Monsters and Mixtures	321
11.1 Ordered categorical outcomes	321
11.2 Zero-inflated outcomes	338
11.3 Over-dispersed outcomes	343
Reference	354
Session info	354
12 Multilevel Models	357
12.1 Example: Multilevel tadpoles	357
12.2 Varying effects and the underfitting/overfitting trade-off	363
12.3 More than one type of cluster	373
12.4 Multilevel posterior predictions	380
12.5 Summary Bonus: <code>tidybayes::spread_draws()</code>	395
Reference	408
Session info	408
13 Adventures in Covariance	411
13.1 Varying slopes by construction	411
13.2 Example: Admission decisions and gender	421
13.3 Example: Cross-classified <code>chimpanzees</code> with varying slopes	431
13.4 Continuous categories and the Gaussian process	435
13.5 Summary Bonus: Another Berkley-admissions-data-like example.	446
Reference	449
Session info	450
14 Missing Data and Other Opportunities	451
14.1 Measurement error	454
14.2 Missing data	468
14.3 Summary Bonus: Meta-analysis	478
Reference	486
Session info	486

15 Horosceopes Insights	489
15.1 Use R Notebooks	489
15.2 Save your model fits	489
15.3 Build your models slowly	492
15.4 Look at your data	494
15.5 Use the <code>0 + intercept</code> syntax	494
15.6 Annotate your workflow	495
15.7 Annotate your code	495
15.8 Break up your workflow	495
15.9 Read Gelman's blog	495
15.10 Check out other social media, too	496
15.11 Parting wisdom	496
Reference	496
Session info	497

This is a love letter

I love McElreath's [Statistical Rethinking text](#). It's the entry-level textbook for applied researchers I spent years looking for. McElreath's [freely-available lectures](#) on the book are really great, too.

However, I prefer using Bürkner's [brms package](#) when doing Bayesian regression in R. It's just spectacular. I also prefer plotting with Wickham's [ggplot2](#), and coding with functions and principles from the [tidyverse](#), which you might learn about [here](#) or [here](#).

So, this project is an attempt to reexpress the code in McElreath's textbook. His models are re-fit with brms, the figures are reproduced or reimaged with ggplot2, and the general data wrangling code now predominantly follows the tidyverse style.

Why this?

I'm not a statistician and I have no formal background in computer science. Though I benefited from a suite of statistics courses in grad school, a large portion of my training has been outside of the classroom, working with messy real-world data, and searching online for help. One of the great resources I happened on was [idre, the UCLA Institute for Digital Education](#), which offers an online portfolio of [richly annotated textbook examples](#). Their online tutorials are among the earliest inspirations for this project. We need more resources like them.

With that in mind, one of the strengths of McElreath's text is its thorough integration with the [rethinking package](#). The rethinking package is a part of the R ecosystem, which is great because R is free and open source. And McElreath has made the source code for rethinking [publically available](#), too. Since he completed his text, [many other packages have been developed](#) to help users of the R ecosystem interface with [Stan](#). Of those alternative packages, I think Bürkner's [brms](#) is the best for general-purpose Bayesian data analysis. It's flexible, uses reasonably-approachable syntax, has sensible defaults, and offers a vast array of post-processing convenience functions. And brms has only gotten [better over time](#). To my knowledge, there are no textbooks on the market that highlight the brms package, which seems like an evil worth correcting.

In addition, McElreath's data wrangling code is based in the base R style and he made most of his figures with base R plots. Though there are benefits to sticking close to base R functions (e.g., less dependencies leading to a lower likelihood that your code will break in the future), there are downsides. [For beginners, base R functions can be difficult both to learn and to read](#). Happily, in recent years Hadley Wickham and others have been developing a group of packages collectively called the [tidyverse](#). These tidyverse packages (e.g., [dplyr](#), [tidyr](#), [purrr](#)) were developed according to an [underlying philosophy](#) and they are designed to work together coherently and seamlessly. Though [not all](#) within the R community share this opinion, I am among those who think the tidyverse style of coding is generally [easier to learn and sufficiently powerful](#) that these packages can accommodate the bulk of your data needs. I also find tidyverse-style syntax easier to read. And of course, the widely-used [ggplot2 package](#) is part of the tidyverse, too.

To be clear, students can get a great education in both Bayesian statistics and programming in R with McElreath's text just the way it is. Just go slow, work through all the examples, and read the text closely. It's a pedagogical boon. I could not have done better or even closely so. But what I can offer is a parallel introduction on how to fit the statistical models with the ever-improving and already-quite-impressive brms package. I can throw in examples of how to perform other operations according to the ethic of the tidyverse. And I can also offer glimpses of some of the other great packages in the R + Stan ecosystem, such as [loo](#), [bayesplot](#), and [tidybayes](#).

My assumptions about you

If you're looking at this project, I'm guessing you're either a graduate student, a post-graduate academic, or a researcher of some sort. So I'm presuming you have at least a 101-level foundation in statistics. If you're rusty, consider checking out Legler and Roback's free bookdown text, [Broadening Your Statistical Horizons](#) before diving into *Statistical Rethinking*. I'm

also assuming you understand the rudiments of R and have at least a vague idea about what the tidyverse is. If you're totally new to R, consider starting with Peng's *R Programming for Data Science*. And the best introduction to the tidyverse-style of data analysis I've found is Grolemund and Wickham's *R for Data Science*, which I extensively link to throughout this project.

That said, you do not need to be totally fluent in statistics or R. Otherwise why would you need this project, anyway? IMO, the most important things are curiosity, a willingness to try, and persistent tinkering. I love this stuff. Hopefully you will, too.

How to use and understand this project

This project is not meant to stand alone. It's a supplement to McElreath's *Statistical Rethinking* text. I follow the structure of his text, chapter by chapter, translating his analyses into brms and tidyverse code. However, some of the sections in the text are composed entirely of equations and prose, leaving us nothing to translate. When we run into those sections, the corresponding sections in this project will sometimes be blank or omitted, though I do highlight some of the important points in quotes and prose of my own. So I imagine students might reference this project as they progress through McElreath's text. I also imagine working data analysts might use this project in conjunction with the text as they flip to the specific sections that seem relevant to solving their data challenges.

I reproduce the bulk of the figures in the text, too. The plots in the first few chapters are the closest to those in the text. However, I'm passionate about data visualization and like to play around with [color palettes](#), formatting templates, and other conventions quite a bit. As a result, the plots in each chapter have their own look and feel. For more on some of these topics, check out chapters 3, 7, and 28 in *R4DS*, Healy's *Data Visualization: A practical introduction*, or Wilke's *Fundamentals of Data Visualization*.

In this project, I use a handful of formatting conventions gleaned from *R4DS*, *The tidyverse style guide*, and *R Markdown: The Definitive Guide*.

- R code blocks and their output appear in a gray background. E.g.,

```
2 + 2 == 5
```

```
## [1] FALSE
```

- Functions are in a typewriter font and followed by parentheses, all atop a gray background (e.g., `brm()`).
- When I want to make explicit the package a given function comes from, I insert the double-colon operator `::` between the package name and the function (e.g., `tidybayes::mode_hdi()`).
- R objects, such as data or function arguments, are in typewriter font atop gray backgrounds (e.g., `chimpanzees`, `.width = .5`).
- You can detect hyperlinks by their typical [blue-colored font](#).
- In the text, McElreath indexed his models with names like `m4.1` (i.e., the first model of Chapter 4). I primarily followed that convention, but replaced the `m` with a `b` to stand for the brms package.

You can do this, too

This project is powered by Yihui Xie's [bookdown package](#), which makes it easy to turn R markdown files into HTML, PDF, and EPUB. Go [here](#) to learn more about bookdown. While you're at it, also check out Xie, Allaire, and Grolemund's *R Markdown: The Definitive Guide*. And if you're unacquainted with GitHub, check out Jenny Bryan's [Happy Git and GitHub for the userR](#). I've even [blogged](#) about what it was like putting together the first version of this project.

The source code of the project is available [here](#).

We have updates

I released the initial 0.9.0 version of this project in September 26, 2018. In April 19, 2019 came the 1.0.0 version. Some of the major changes were:

- All models were refit with the current official version of brms, 2.8.0.
- Adopting the `seed` argument within the `brm()` function made the model results more reproducible.
- The `loo` package was updated. As a consequence, our workflow for the WAIC and LOO changed, too.
- I improved the brms alternative to McElreath’s `coeftab()` function.
- I made better use of the tidyverse, especially some of the `purrr` functions.
- Particularly in the later chapters, there’s a greater emphasis on functions from the `tidybayes` package.
- Chapter 11 contains the updated brms 2.8.0 workflow for making custom distributions, using the beta-binomial model as the example.
- Chapter 12 received a new bonus section contrasting different methods for working with multilevel posteriors.
- Chapter 14 received a new bonus section introducing Bayesian meta-analysis and linking it to multilevel and measurement-error models.
- With the help of others within the community, I corrected many typos and streamlined some of the code (e.g., [dropped an unnecessary use of the mi\(\) function in section 14.2.1](#))
- And in some cases, I corrected sections that were just plain wrong (e.g., some of my initial attempts in section 3.3 were incorrect).

In response to some reader requests, we finally have a PDF version! Making that happen required some formatting adjustments, resulting in version 1.0.1. Noteworthy changes include:

- Major revisions to the LaTeX syntax underlying many of the in-text equations (e.g., dropping the “eqnarray” environment for “align*”)
- Adjusting some of the image syntax
- Updating the reference for the Bayesian R^2 ([Gelman, Goodrich, Gabry, & Vehtari, 2018](#))

Though we’re into version 1.0.1, there’s room for improvement. There are still two models that need work. The current solution for model 10.6 is [wrong](#), which I try to make clear in the prose. It also appears that the Gaussian process model from section 13.4 is off. Both models are beyond my current skill set and [friendly suggestions are welcome](#). In addition to modeling concerns, typos may yet be looming and I’m sure there are places where the code could be made more streamlined, more elegant, or just more in-line with the tidyverse style. Which is all to say, I hope to release better and more useful updates in the future.

Before we move on, I’d like to thank the following for their helpful contributions: Paul-Christian Bürkner ([@paul-buerkner](#)), Andrew Collier ([@datawookie](#)), Jeff Hammerbacher ([@hammer](#)), Matthew Kay ([@mjskay](#)), TJ Mahr ([@tjmahr](#)), Stijn Masschelein ([@stijnmasschelein](#)), Colin Quirk ([@colinquirk](#)), Rishi Sadhir ([@RishiSadhir](#)), Richard Torkar ([@torkar](#)), Aki Vehtari ([@avehtari](#)).

Chapter 1

The Golem of Prague

As he opened the chapter, McElreath told us that

ultimately Judah was forced to destroy the golem, as its combination of extraordinary power with clumsiness eventually led to innocent deaths. Wiping away one letter from the inscription *emet* to spell instead *met*, “death,” Rabbi Judah decommissioned the robot.

1.1 Statistical golems

Scientists also make golems. Our golems rarely have physical form, but they too are often made of clay, living in silicon as computer code. These golems are scientific model. But these golems have real effects on the world, through the predictions they make and the intuitions they challenge or inspire. A concern with truth enlivens these models, but just like a golem or a modern robot, scientific models are neither true nor false, neither prophets nor charlatans. Rather they are constructs engineered for some purpose. These constructs are incredibly powerful, dutifully conducting their programmed calculations. (p. 1, *emphasis* in the original)

There are a lot of great points, themes, methods, and factoids in this text. For me, one of the most powerful themes interlaced throughout the pages is how we should be skeptical of our models. Yes, learn Bayes. Pour over this book. Fit models until late into the night. But please don’t fall into blind love with their elegance and power. If we all knew what we were doing, there’d be no need for science. For more wise deflation along these lines, do check out [A personal essay on Bayes factors, Between the Devil and the Deep Blue Sea: Tensions Between Scientific Judgement and Statistical Model Selection and Science, statistics and the problem of “pretty good inference”](#), a blog, paper and talk by the inimitable Danielle Navarro.



Figure 1.1: Rabbi Loew and Golem by Mikoláš Aleš, 1899

Anyway, McElreath left us no code or figures to translate in this chapter. But before you skip off to the next one, why not invest a little time soaking in this chapter's material by watching [McElreath present it?](#) He's an engaging speaker and the material in his online lectures does not entirely overlap with that in the text.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats      graphics   grDevices  utils      datasets   methods    base
##
## loaded via a namespace (and not attached):
## [1] compiler_3.5.1  backports_1.1.4  bookdown_0.9    magrittr_1.5
## [5] rprojroot_1.3-2 tools_3.5.1     htmltools_0.3.6  yaml_2.1.19
## [9] Rcpp_1.0.1      stringi_1.4.3   rmarkdown_1.10  knitr_1.20
## [13] xfun_0.3       stringr_1.4.0   digest_0.6.18   evaluate_0.10.1
```

Chapter 2

Small Worlds and Large Worlds

A while back The Oatmeal put together an [infographic on Christopher Columbus](#). I'm no historian and cannot vouch for its accuracy, so make of it what you will.

McElreath described the thrust of this chapter this way:

In this chapter, you will begin to build Bayesian models. The way that Bayesian models learn from evidence is arguably optimal in the small world. When their assumptions approximate reality, they also perform well in the large world. But large world performance has to be demonstrated rather than logically deduced. (p. 20)

Indeed.

2.1 The garden of forking data

Gelman and Loken wrote a [great paper by this name](#).

2.1.1 Counting possibilities.

Throughout this project, we'll use the [tidyverse](#) for data wrangling.

```
library(tidyverse)
```

If you are new to tidyverse-style syntax, possibly the oddest component is the pipe (i.e., `%>%`). I'm not going to explain the `%>%` in this project, but you might learn more about in [this brief clip](#), starting around [minute 21:25 in this talk by Wickham](#), or in [section 5.6.1 from Golemund and Wickham's R for Data Science](#). Really, all of Chapter 5 of *R4DS* is just great for new R and new tidyverse users. And *R4DS* Chapter 3 is a nice introduction to plotting with ggplot2.

Other than the pipe, the other big thing to be aware of is [tibbles](#). For our purposes, think of a tibble as a data object with two dimensions defined by rows and columns. And importantly, tibbles are just special types of [data frames](#). So whenever we talk about data frames, we're also talking about tibbles. For more on the topic, check out [R4SD, Chapter 10](#).

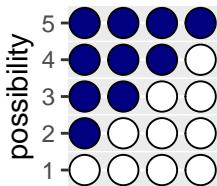
So, if we're willing to code the marbles as 0 = "white" 1 = "blue", we can arrange the possibility data in a tibble as follows.

```
d <-  
  tibble(p_1 = 0,  
         p_2 = rep(1:0, times = c(1, 3)),  
         p_3 = rep(1:0, times = c(2, 2)),  
         p_4 = rep(1:0, times = c(3, 1)),  
         p_5 = 1)  
  
head(d)
```

```
## # A tibble: 4 x 5
##   p_1   p_2   p_3   p_4   p_5
##   <dbl> <int> <int> <int> <dbl>
## 1     0     1     1     1     1
## 2     0     0     1     1     1
## 3     0     0     0     1     1
## 4     0     0     0     0     1
```

You might depict the possibility data in a plot.

```
d %>%
  gather() %>%
  mutate(x = rep(1:4, times = 5),
        possibility = rep(1:5, each = 4)) %>%
  ggplot(aes(x = x, y = possibility,
             fill = value %>% as.character())) +
  geom_point(shape = 21, size = 5) +
  scale_fill_manual(values = c("white", "navy")) +
  scale_x_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = c(.75, 4.25),
                  ylim = c(.75, 5.25)) +
  theme(legend.position = "none")
```



As a quick aside, check out Suzan Baert's blog post [Data Wrangling Part 2: Transforming your columns into the right shape](#) for an extensive discussion on `dplyr::mutate()` and `dplyr::gather()`.

Here's the basic structure of the possibilities per marble draw.

```
tibble(draw      = 1:3,
       marbles    = 4) %>%
  mutate(possibilities = marbles ^ draw) %>%
  knitr::kable()
```

draw	marbles	possibilities
1	4	4
2	4	16
3	4	64

If you walk that out a little, you can structure the data required to approach Figure 2.2.

```
(

d <-
tibble(position = c((1:4^1) / 4^0,
                    (1:4^2) / 4^1,
                    (1:4^3) / 4^2),
       draw      = rep(1:3, times = c(4^1, 4^2, 4^3)),
       fill      = rep(c("b", "w"), times = c(1, 3)) %>%
                    rep(., times = c(4^0 + 4^1 + 4^2)))
)

## # A tibble: 84 x 3
##   position  draw  fill
```

```

##      <dbl> <int> <chr>
## 1     1       1   b
## 2     2       1   w
## 3     3       1   w
## 4     4       1   w
## 5    0.25     2   b
## 6    0.5      2   w
## 7    0.75     2   w
## 8     1       2   w
## 9   1.25     2   b
## 10   1.5      2   w
## # ... with 74 more rows

```

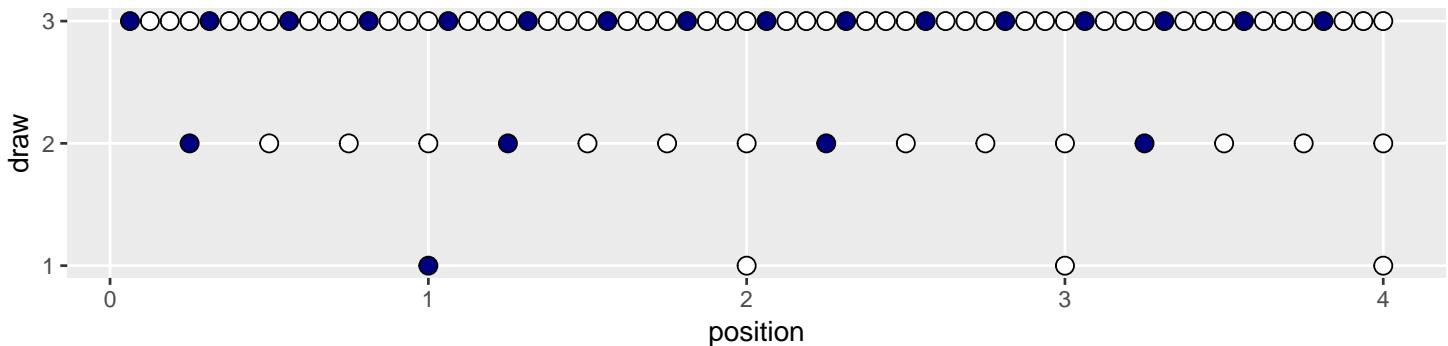
See what I did there with the parentheses? If you assign a value to an object in R (e.g., `dog <- 1`) and just hit return, nothing will immediately pop up in the [console](#). You have to actually execute `dog` before R will return 1. But if you wrap the code within parentheses (e.g., `(dog <- 1)`), R will perform the assignment and return the value as if you had executed `dog`.

But we digress. Here's the initial plot.

```

d %>%
  ggplot(aes(x = position, y = draw)) +
  geom_point(aes(fill = fill),
             shape = 21, size = 3) +
  scale_y_continuous(breaks = 1:3) +
  scale_fill_manual(values = c("navy", "white")) +
  theme(panel.grid.minor = element_blank(),
        legend.position = "none")

```



To my mind, the easiest way to connect the dots in the appropriate way is to make two auxiliary tibbles.

```

# these will connect the dots from the first and second draws
(
  lines_1 <-
  tibble(x      = rep((1:4), each = 4),
        xend = ((1:4^2) / 4),
        y      = 1,
        yend = 2)
)

```

```

## # A tibble: 16 x 4
##      x   xend     y   yend
##   <int> <dbl> <dbl> <dbl>
## 1     1   0.25     1     2
## 2     1   0.5      1     2
## 3     1   0.75     1     2
## 4     1   1         1     2
## 5     2   1.25     1     2

```

```
## 6   2  1.5    1   2
## 7   2  1.75   1   2
## 8   2  2      1   2
## 9   3  2.25   1   2
## 10  3  2.5    1   2
## 11  3  2.75   1   2
## 12  3  3      1   2
## 13  4  3.25   1   2
## 14  4  3.5    1   2
## 15  4  3.75   1   2
## 16  4  4      1   2
```

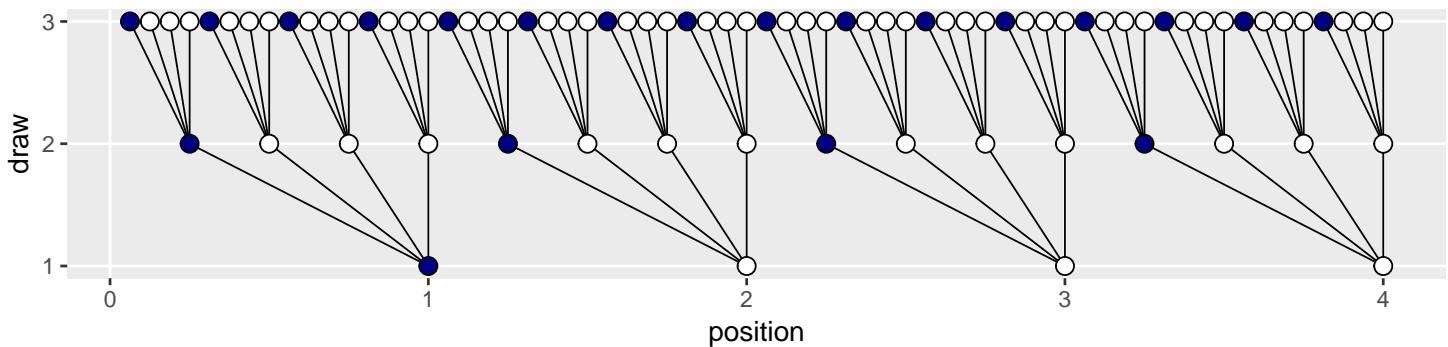
these will connect the dots from the second and third draws

```
(  
  lines_2 <-  
  tibble(x = rep(((1:4^2) / 4), each = 4),  
         xend = (1:4^3) / (4^2),  
         y = 2,  
         yend = 3)  
)
```

```
## # A tibble: 64 x 4  
##       x     xend     y     yend  
##   <dbl>   <dbl> <dbl> <dbl>  
## 1  0.25  0.0625    2     3  
## 2  0.25  0.125     2     3  
## 3  0.25  0.188     2     3  
## 4  0.25  0.25      2     3  
## 5  0.5   0.312     2     3  
## 6  0.5   0.375     2     3  
## 7  0.5   0.438     2     3  
## 8  0.5   0.5       2     3  
## 9  0.75  0.562     2     3  
## 10 0.75  0.625     2     3  
## # ... with 54 more rows
```

We can use the `lines_1` and `lines_2` data in the plot with two `geom_segment()` functions.

```
d %>%  
  ggplot(aes(x = position, y = draw)) +  
  geom_segment(data = lines_1,  
               aes(x = x, xend = xend,  
                    y = y, yend = yend),  
               size = 1/3) +  
  geom_segment(data = lines_2,  
               aes(x = x, xend = xend,  
                    y = y, yend = yend),  
               size = 1/3) +  
  geom_point(aes(fill = fill),  
             shape = 21, size = 3) +  
  scale_y_continuous(breaks = 1:3) +  
  scale_fill_manual(values = c("navy", "white")) +  
  theme(panel.grid.minor = element_blank(),  
        legend.position = "none")
```



We've generated the values for `position` (i.e., the x-axis), in such a way that they're all justified to the right, so to speak. But we'd like to center them. For `draw == 1`, we'll need to subtract 0.5 from each. For `draw == 2`, we need to reduce the scale by a factor of 4 and we'll then need to reduce the scale by another factor of 4 for `draw == 3`. The `ifelse()` function will be of use for that.

```
d <-
d %>%
  mutate(denominator = ifelse(draw == 1, .5,
                               ifelse(draw == 2, .5 / 4,
                                     .5 / 4^2))) %>%
  mutate(position      = position - denominator)

d
```

```
## # A tibble: 84 x 4
##   position draw fill  denominator
##       <dbl> <int> <chr>     <dbl>
## 1     0.5     1 b         0.5
## 2     1.5     1 w         0.5
## 3     2.5     1 w         0.5
## 4     3.5     1 w         0.5
## 5     0.125    2 b        0.125
## 6     0.375    2 w        0.125
## 7     0.625    2 w        0.125
## 8     0.875    2 w        0.125
## 9     1.12     2 b        0.125
## 10    1.38     2 w        0.125
## # ... with 74 more rows
```

We'll follow the same logic for the `lines_1` and `lines_2` data.

```
(
  lines_1 <-
  lines_1 %>%
  mutate(x      = x - .5,
        xend = xend - .5 / 4^1)
)
```

```
## # A tibble: 16 x 4
##   x xend     y yend
##   <dbl> <dbl> <dbl> <dbl>
## 1 0.5 0.125  1    2
## 2 0.5 0.375  1    2
## 3 0.5 0.625  1    2
## 4 0.5 0.875  1    2
## 5 1.5 1.12   1    2
## 6 1.5 1.38   1    2
## 7 1.5 1.62   1    2
```

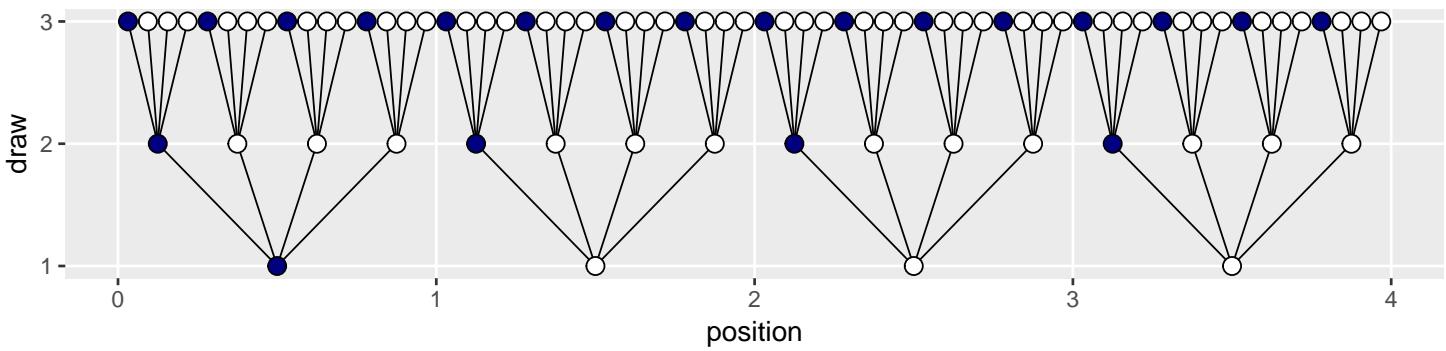
```
##  8  1.5 1.88      1      2
##  9  2.5 2.12      1      2
## 10  2.5 2.38      1      2
## 11  2.5 2.62      1      2
## 12  2.5 2.88      1      2
## 13  3.5 3.12      1      2
## 14  3.5 3.38      1      2
## 15  3.5 3.62      1      2
## 16  3.5 3.88      1      2
```

```
(  
  lines_2 <-  
  lines_2 %>%  
  mutate(x    = x - .5 / 4^1,  
        xend = xend - .5 / 4^2)  
)
```

```
## # A tibble: 64 x 4  
##       x     xend     y     yend  
##   <dbl>   <dbl> <dbl> <dbl>  
## 1 0.125 0.0312  2     3  
## 2 0.125 0.0938  2     3  
## 3 0.125 0.156   2     3  
## 4 0.125 0.219   2     3  
## 5 0.375 0.281   2     3  
## 6 0.375 0.344   2     3  
## 7 0.375 0.406   2     3  
## 8 0.375 0.469   2     3  
## 9 0.625 0.531   2     3  
## 10 0.625 0.594  2     3  
## # ... with 54 more rows
```

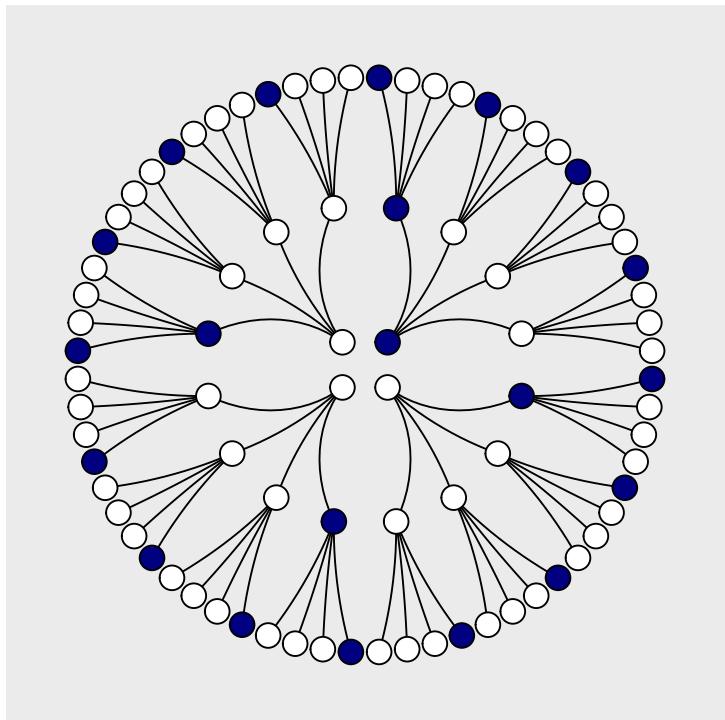
Now the plot's looking closer.

```
d %>%  
  ggplot(aes(x = position, y = draw)) +  
  geom_segment(data = lines_1,  
               aes(x = x, xend = xend,  
                    y = y, yend = yend),  
               size = 1/3) +  
  geom_segment(data = lines_2,  
               aes(x = x, xend = xend,  
                    y = y, yend = yend),  
               size = 1/3) +  
  geom_point(aes(fill = fill),  
             shape = 21, size = 3) +  
  scale_y_continuous(breaks = 1:3) +  
  scale_fill_manual(values = c("navy", "white")) +  
  theme(panel.grid.minor = element_blank(),  
        legend.position = "none")
```



For the final step, we'll use `coord_polar()` to change the coordinate system, giving the plot a mandala-like feel.

```
d %>%
  ggplot(aes(x = position, y = draw)) +
  geom_segment(data = lines_1,
    aes(x = x, xend = xend,
        y = y, yend = yend),
    size = 1/3) +
  geom_segment(data = lines_2,
    aes(x = x, xend = xend,
        y = y, yend = yend),
    size = 1/3) +
  geom_point(aes(fill = fill),
    shape = 21, size = 4) +
  scale_fill_manual(values = c("navy", "white")) +
  scale_x_continuous(NULL, limits = c(0, 4), breaks = NULL) +
  scale_y_continuous(NULL, limits = c(0.75, 3), breaks = NULL) +
  theme(panel.grid = element_blank(),
    legend.position = "none") +
  coord_polar()
```



To make our version of Figure 2.3, we'll have to add an index to tell us which paths remain logically valid after each choice. We'll call the index `remain`.

```

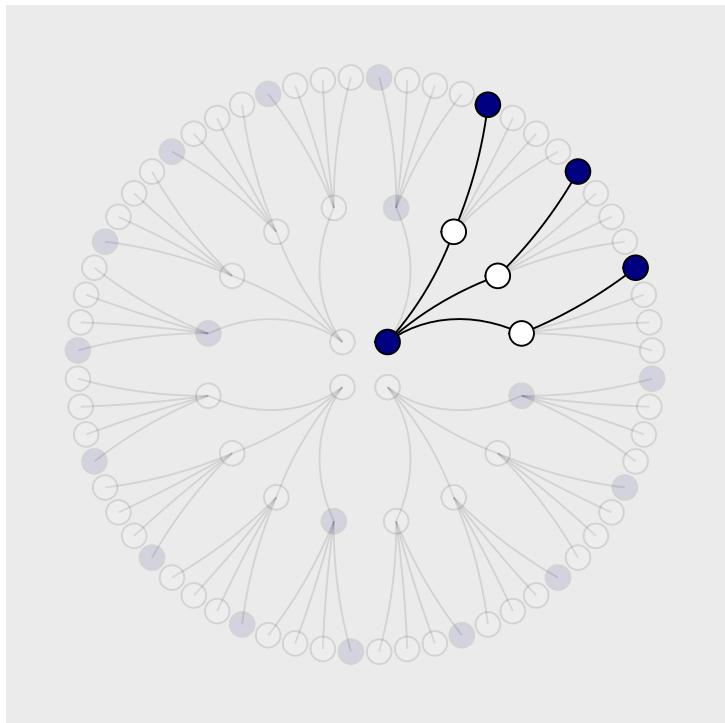
lines_1 <-
  lines_1 %>%
  mutate(remain = c(rep(0:1, times = c(1, 3)),
    rep(0,   times = 4 * 3)))

lines_2 <-
  lines_2 %>%
  mutate(remain = c(rep(0,   times = 4),
    rep(1:0, times = c(1, 3)) %>%
      rep(., times = 3),
    rep(0,   times = 12 * 4)))

d <-
  d %>%
  mutate(remain = c(rep(1:0, times = c(1, 3)),
    rep(0:1, times = c(1, 3)),
    rep(0,   times = 4 * 4),
    rep(1:0, times = c(1, 3)) %>%
      rep(., times = 3),
    rep(0,   times = 12 * 4)))

# finally, the plot:
d %>%
  ggplot(aes(x = position, y = draw)) +
  geom_segment(data = lines_1,
    aes(x = x, xend = xend,
        y = y, yend = yend,
        alpha = remain %>% as.character(),
        size = 1/3) +
  geom_segment(data = lines_2,
    aes(x = x, xend = xend,
        y = y, yend = yend,
        alpha = remain %>% as.character(),
        size = 1/3) +
  geom_point(aes(fill = fill, alpha = remain %>% as.character()),
    shape = 21, size = 4) +
# it's the alpha parameter that makes elements semitransparent
  scale_alpha_manual(values = c(1/10, 1)) +
  scale_fill_manual(values = c("navy", "white")) +
  scale_x_continuous(NULL, limits = c(0, 4), breaks = NULL) +
  scale_y_continuous(NULL, limits = c(0.75, 3), breaks = NULL) +
  theme(panel.grid      = element_blank(),
    legend.position = "none") +
  coord_polar()

```



Letting “w” = a white dot and “b” = a blue dot, we might recreate the table in the middle of page 23 like so.

```
# if we make two custom functions, here, it will simplify the code within `mutate()`, below
n_blue <- function(x){
  rowSums(x == "b")
}

n_white <- function(x){
  rowSums(x == "w")
}

t <-
  # for the first four columns, `p_` indexes position
  tibble(p_1 = rep(c("w", "b"), times = c(1, 4)),
         p_2 = rep(c("w", "b"), times = c(2, 3)),
         p_3 = rep(c("w", "b"), times = c(3, 2)),
         p_4 = rep(c("w", "b"), times = c(4, 1))) %>%
  mutate(`draw 1: blue` = n_blue(.),
        `draw 2: white` = n_white(.),
        `draw 3: blue` = n_blue(.)) %>%
  mutate(`ways to produce` = `draw 1: blue` * `draw 2: white` * `draw 3: blue`)

t %>%
  knitr::kable()
```

p_1	p_2	p_3	p_4	draw 1: blue	draw 2: white	draw 3: blue	ways to produce
w	w	w	w	0	4	0	0
b	w	w	w	1	3	1	3
b	b	w	w	2	2	2	8
b	b	b	w	3	1	3	9
b	b	b	b	4	0	4	0

We'll need new data for Figure 2.4. Here's the initial primary data, d.

```
d <-
  tibble(position = c((1:4^1) / 4^0,
                      (1:4^2) / 4^1,
```

```

        (1:4^3) / 4^2),
draw      = rep(1:3, times = c(4^1, 4^2, 4^3)))

(
d <-
d %>%
bind_rows(
d, d
) %>%
# here are the fill colors
mutate(fill = c(rep(c("w", "b"), times = c(1, 3)) %>% rep(., times = c(4^0 + 4^1 + 4^2)),
rep(c("w", "b"), each = 2) %>% rep(., times = c(4^0 + 4^1 + 4^2)),
rep(c("w", "b"), times = c(3, 1)) %>% rep(., times = c(4^0 + 4^1 + 4^2)))) %>%
# now we need to shift the positions over in accordance with draw, like before
mutate(denominator = ifelse(draw == 1, .5,
ifelse(draw == 2, .5 / 4,
.5 / 4^2))) %>%
mutate(position = position - denominator) %>%
# here we'll add an index for which pie wedge we're working with
mutate(pie_index = rep(letters[1:3], each = n() / 3)) %>%
# to get the position axis correct for pie_index == "b" or "c", we'll need to offset
mutate(position = ifelse(pie_index == "a", position,
ifelse(pie_index == "b", position + 4,
position + 4 * 2)))
)

## # A tibble: 252 x 5
##   position  draw fill  denominator pie_index
##       <dbl> <int> <chr>      <dbl> <chr>
## 1     0.5     1 w         0.5     a
## 2     1.5     1 b         0.5     a
## 3     2.5     1 b         0.5     a
## 4     3.5     1 b         0.5     a
## 5     0.125    2 w        0.125    a
## 6     0.375    2 b        0.125    a
## 7     0.625    2 b        0.125    a
## 8     0.875    2 b        0.125    a
## 9     1.12     2 w        0.125    a
## 10    1.38     2 b        0.125    a
## # ... with 242 more rows

```

Both `lines_1` and `lines_2` require adjustments for `x` and `xend`. Our current approach is a nested `ifelse()`. Rather than copy and paste that multi-line `ifelse()` code for all four, let's wrap it in a compact function, which we'll call `move_over()`.

```

move_over <- function(position, index){
  ifelse(index == "a", position,
  ifelse(index == "b", position + 4,
  position + 4 * 2)
)
}

```

If you're new to making your own R functions, check out [Chapter 19](#) of *R4DS* or [Chapter 14](#) of *R Programming for Data Science*.

Anyway, now we'll make our new `lines_1` and `lines_2` data, for which we'll use `move_over()` to adjust their `x` and `xend` positions to the correct spots.

```

(
  lines_1 <-
  tibble(x      = rep((1:4), each = 4) %>% rep(., times = 3),
         xend = ((1:4^2) / 4)           %>% rep(., times = 3),
         y      = 1,
         yend = 2) %>%
  mutate(x      = x - .5,
         xend = xend - .5 / 4^1) %>%
  # here we'll add an index for which pie wedge we're working with
  mutate(pie_index = rep(letters[1:3], each = n()/3)) %>%
  # to get the position axis correct for `pie_index == "b"` or `c`, we'll need to offset
  mutate(x      = move_over(position = x,      index = pie_index),
         xend = move_over(position = xend, index = pie_index))
)

## # A tibble: 48 x 5
##       x     xend     y     yend pie_index
##   <dbl>   <dbl>   <dbl>   <dbl>   <chr>
## 1  0.5  0.125    1     2     a
## 2  0.5  0.375    1     2     a
## 3  0.5  0.625    1     2     a
## 4  0.5  0.875    1     2     a
## 5  1.5  1.12     1     2     a
## 6  1.5  1.38     1     2     a
## 7  1.5  1.62     1     2     a
## 8  1.5  1.88     1     2     a
## 9  2.5  2.12     1     2     a
## 10 2.5  2.38     1     2     a
## # ... with 38 more rows

(
  lines_2 <-
  tibble(x      = rep(((1:4^2) / 4), each = 4) %>% rep(., times = 3),
         xend = (1:4^3 / 4^2)           %>% rep(., times = 3),
         y      = 2,
         yend = 3) %>%
  mutate(x      = x - .5 / 4^1,
         xend = xend - .5 / 4^2) %>%
  # here we'll add an index for which pie wedge we're working with
  mutate(pie_index = rep(letters[1:3], each = n()/3)) %>%
  # to get the position axis correct for `pie_index == "b"` or `c`, we'll need to offset
  mutate(x      = move_over(position = x,      index = pie_index),
         xend = move_over(position = xend, index = pie_index))
)

## # A tibble: 192 x 5
##       x     xend     y     yend pie_index
##   <dbl>   <dbl>   <dbl>   <dbl>   <chr>
## 1  0.125  0.0312    2     3     a
## 2  0.125  0.0938    2     3     a
## 3  0.125  0.156     2     3     a
## 4  0.125  0.219     2     3     a
## 5  0.375  0.281     2     3     a
## 6  0.375  0.344     2     3     a
## 7  0.375  0.406     2     3     a
## 8  0.375  0.469     2     3     a
## 9  0.625  0.531     2     3     a
## 10 0.625  0.594     2     3     a
## # ... with 182 more rows

```

For the last data wrangling step, we add the `remain` indices to help us determine which parts to make semitransparent. I'm not sure of a slick way to do this, so these are the result of brute force counting.

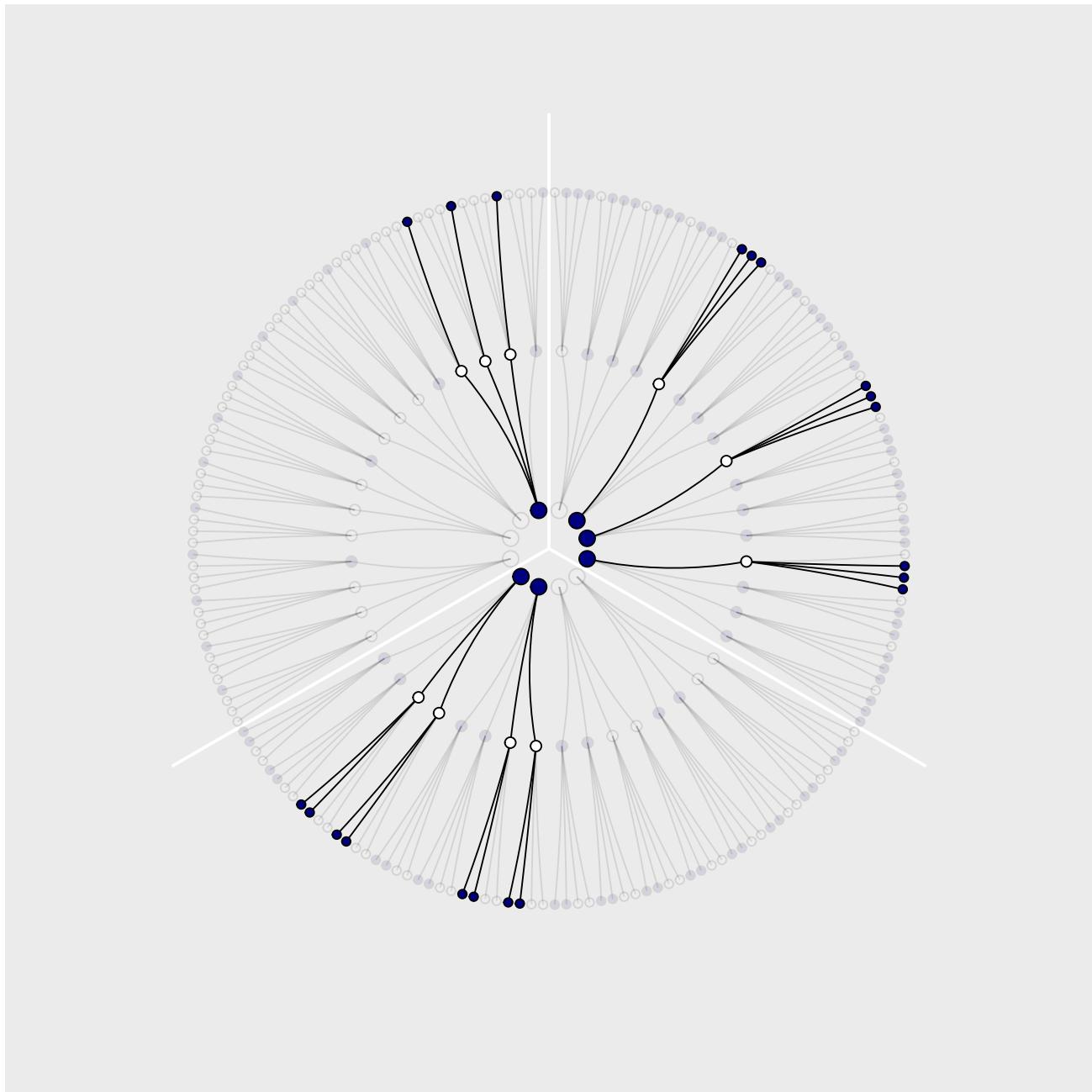
```
d <-
d %>%
  mutate(remain = c(# `pie_index == "a"`
    rep(0:1, times = c(1, 3)),
    rep(0, times = 4),
    rep(1:0, times = c(1, 3)) %>%
      rep(., times = 3),
    rep(0, times = 4 * 4),
    rep(c(0, 1, 0), times = c(1, 3, 4 * 3)) %>%
      rep(., times = 3),
    # `pie_index == "b"`
    rep(0:1, each = 2),
    rep(0, times = 4 * 2),
    rep(1:0, each = 2) %>%
      rep(., times = 2),
    rep(0, times = 4 * 4 * 2),
    rep(c(0, 1, 0, 1, 0), times = c(2, 2, 2, 2, 8)) %>%
      rep(., times = 2),
    # `pie_index == "c"`
    rep(0:1, times = c(3, 1)),
    rep(0, times = 4 * 3),
    rep(1:0, times = c(3, 1)),
    rep(0, times = 4 * 4 * 3),
    rep(0:1, times = c(3, 1)) %>%
      rep(., times = 3),
    rep(0, times = 4)
  )
)

lines_1 <-
lines_1 %>%
  mutate(remain = c(rep(0, times = 4),
    rep(1:0, times = c(1, 3)) %>%
      rep(., times = 3),
    rep(0, times = 4 * 2),
    rep(1:0, each = 2) %>%
      rep(., times = 2),
    rep(0, times = 4 * 3),
    rep(1:0, times = c(3, 1))
  )
)

lines_2 <-
lines_2 %>%
  mutate(remain = c(rep(0, times = 4 * 4),
    rep(c(0, 1, 0), times = c(1, 3, 4 * 3)) %>%
      rep(., times = 3),
    rep(0, times = 4 * 8),
    rep(c(0, 1, 0, 1, 0), times = c(2, 2, 2, 2, 8)) %>%
      rep(., times = 2),
    rep(0, times = 4 * 4 * 3),
    rep(0:1, times = c(3, 1)) %>%
      rep(., times = 3),
    rep(0, times = 4)
  )
)
```

We're finally ready to plot our Figure 2.4.

```
d %>%
  ggplot(aes(x = position, y = draw)) +
  geom_vline(xintercept = c(0, 4, 8), color = "white", size = 2/3) +
  geom_segment(data = lines_1,
    aes(x = x, xend = xend,
        y = y, yend = yend,
        alpha = remain %>% as.character(),
        size = 1/3) +
  geom_segment(data = lines_2,
    aes(x = x, xend = xend,
        y = y, yend = yend,
        alpha = remain %>% as.character(),
        size = 1/3) +
  geom_point(aes(fill = fill, size = draw, alpha = remain %>% as.character()),
    shape = 21) +
  scale_size_continuous(range = c(3, 1.5)) +
  scale_alpha_manual(values = c(1/10, 1)) +
  scale_fill_manual(values = c("navy", "white")) +
  scale_x_continuous(NULL, limits = c(0, 12), breaks = NULL) +
  scale_y_continuous(NULL, limits = c(0.75, 3.5), breaks = NULL) +
  theme(panel.grid = element_blank(),
    legend.position = "none") +
  coord_polar()
```



2.1.2 Using prior information.

We may have prior information about the relative plausibility of each conjecture. This prior information could arise from knowledge of how the contents of the bag were generated. It could also arise from previous data. Or we might want to act as if we had prior information, so we can build conservatism into the analysis. Whatever the source, it would help to have a way to use prior information. Luckily there is a natural solution: Just multiply the prior count by the new count. (p. 25)

Here's the table in the middle of page 25.

```
t <-
t %>%
  rename(`previous counts` = `ways to produce`,
         `ways to produce` = `draw 1: blue`) %>%
  select(p_1:p_4, `ways to produce`, `previous counts`) %>%
  mutate(`new count` = `ways to produce` * `previous counts`)
```

```
t %>%
  knitr::kable()
```

p_1	p_2	p_3	p_4	ways to produce	previous counts	new count
w	w	w	w	0	0	0
b	w	w	w	1	3	3
b	b	w	w	2	8	16
b	b	b	w	3	9	27
b	b	b	b	4	0	0

We might update to reproduce the table at the top of page 26, like this.

```
t <-
  t %>%
  select(p_1:p_4, `new count`) %>%
  rename(`prior count` = `new count`) %>%
  mutate(`factory count` = c(0, 3:0)) %>%
  mutate(`new count` = `prior count` * `factory count`)

t %>%
  knitr::kable()
```

p_1	p_2	p_3	p_4	prior count	factory count	new count
w	w	w	w	0	0	0
b	w	w	w	3	3	9
b	b	w	w	16	2	32
b	b	b	w	27	1	27
b	b	b	b	0	0	0

To learn more about `dplyr::select()` and `dplyr::rename()`, check out Baert's exhaustive blog post [Data Wrangling Part 1: Basic to Advanced Ways to Select Columns](#).

2.1.3 From counts to probability.

The opening sentences in this subsection are important: "It is helpful to think of this strategy as adhering to a principle of honest ignorance: *When we don't know what caused the data, potential causes that may produce the data in more ways are more plausible*" (p. 26, *emphasis* in the original).

We can define our updated plausibility as:

plausibility of [●○○○] after seeing ●○●

∞

ways [●○○○] can produce ●○●

×

prior plausibility of [●○○○]

In other words:

$$\text{plausibility of } p \text{ after } D_{\text{new}} \propto \text{ways } p \text{ can produce } D_{\text{new}} \times \text{prior plausibility of } p$$

But since we have to standardize the results to get them into a probability metric, the full equation is:

$$\text{plausibility of } p \text{ after } D_{\text{new}} = \frac{\text{ways } p \text{ can produce } D_{\text{new}} \times \text{prior plausibility of } p}{\text{sum of the products}}$$

You might make the table in the middle of page 27 like this.

```
t %>%
  select(p_1:p_4) %>%
  mutate(p = seq(from = 0, to = 1, by = .25),
    `ways to produce data` = c(0, 3, 8, 9, 0)) %>%
  mutate(plausibility = `ways to produce data` / sum(`ways to produce data`))

## # A tibble: 5 x 7
##   p_1   p_2   p_3   p_4      p `ways to produce data` plausibility
##   <chr> <chr> <chr> <chr> <dbl>                <dbl>            <dbl>
## 1 w     w     w     w     0                  0                 0
## 2 b     w     w     w     0.25              3                 0.15
## 3 b     b     w     w     0.5               8                 0.4
## 4 b     b     b     w     0.75              9                 0.45
## 5 b     b     b     b     1                  0                 0
```

We just computed the plausibilities, but here's McElreath's R code 2.1.

```
ways <- c(0, 3, 8, 9, 0)
ways / sum(ways)

## [1] 0.00 0.15 0.40 0.45 0.00
```

2.2 Building a model

We might save our globe-tossing data in a tibble.

```
(d <- tibble(toss = c("w", "l", "w", "w", "w", "l", "w", "l", "w")))
## # A tibble: 9 x 1
##   toss
##   <chr>
## 1 w
## 2 l
## 3 w
## 4 w
## 5 w
## 6 l
## 7 w
## 8 l
## 9 w
```

2.2.1 A data story.

Bayesian data analysis usually means producing a story for how the data came to be. This story may be *descriptive*, specifying associations that can be used to predict outcomes, given observations. Or it may be *causal*, a theory of how come events produce other events. Typically, any story you intend to be causal may also be descriptive. But many descriptive stories are hard to interpret causally. But all data stories are complete, in the sense that they are sufficient for specifying an algorithm for simulating new data. (p. 28, *emphasis* in the original)

2.2.2 Bayesian updating.

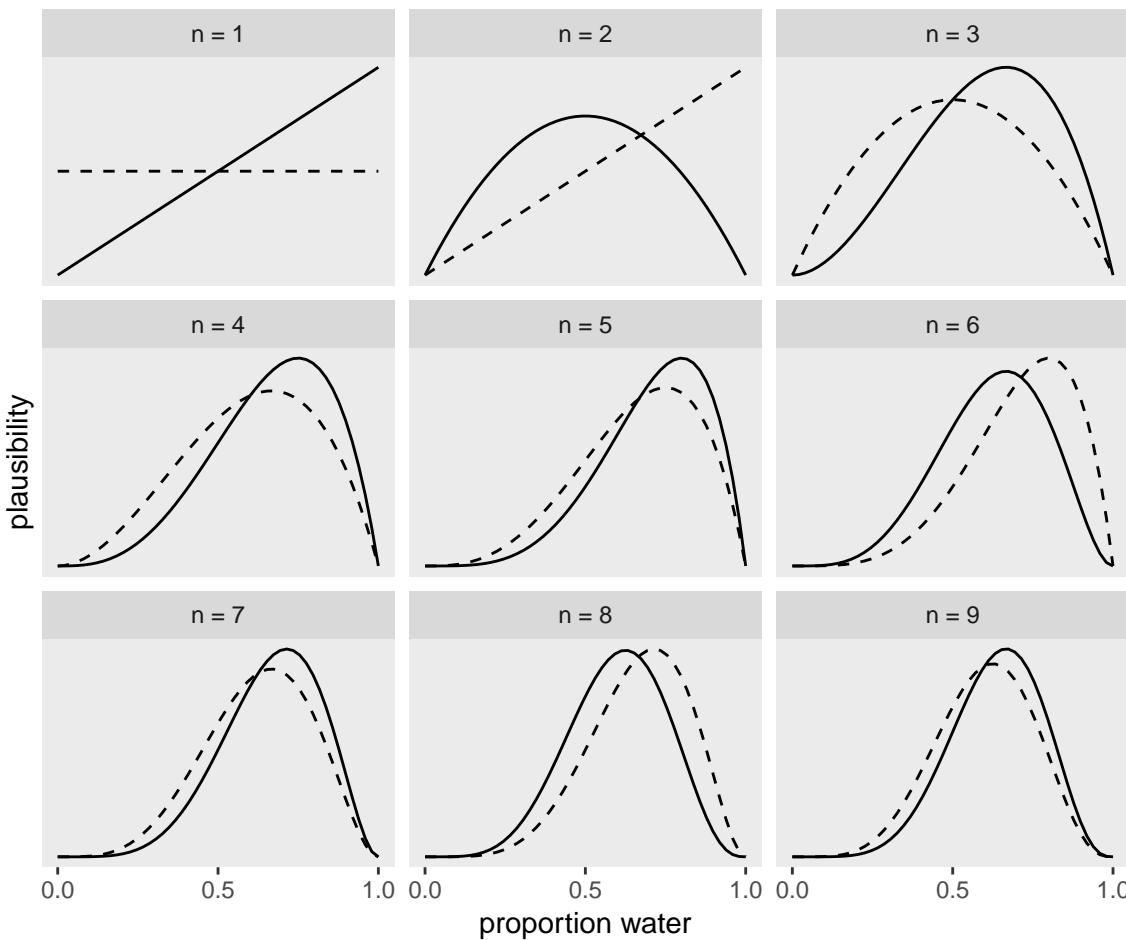
Here we'll add the cumulative number of trials, `n_trials`, and the cumulative number of successes, `n_successes` (i.e., `toss == "w"`), to the data.

```
(  
  d <-  
  d %>%  
  mutate(n_trials = 1:9,  
        n_success = cumsum(toss == "w"))  
)
```

```
## # A tibble: 9 x 3  
##   toss  n_trials n_success  
##   <chr>     <int>      <int>  
## 1 w          1         1  
## 2 l          2         1  
## 3 w          3         2  
## 4 w          4         3  
## 5 w          5         4  
## 6 l          6         4  
## 7 w          7         5  
## 8 l          8         5  
## 9 w          9         6
```

Fair warning: We don't learn the skills for making Figure 2.5 until later in the chapter. So consider the data wrangling steps in this section as something of a preview.

```
sequence_length <- 50  
  
d %>%  
  expand(nesting(n_trials, toss, n_success),  
         p_water = seq(from = 0, to = 1, length.out = sequence_length)) %>%  
  group_by(p_water) %>%  
  # you can learn more about lagging here: https://www.rdocumentation.org/packages/stats/versions/3.5.1/topics/  
  mutate(lagged_n_trials = lag(n_trials, k = 1),  
        lagged_n_success = lag(n_success, k = 1)) %>%  
  ungroup() %>%  
  mutate(prior      = ifelse(n_trials == 1, .5,  
                             dbinom(x      = lagged_n_success,  
                                   size = lagged_n_trials,  
                                   prob = p_water)),  
        likelihood = dbinom(x      = n_success,  
                             size = n_trials,  
                             prob = p_water),  
        strip      = str_c("n = ", n_trials)  
  ) %>%  
  # the next three lines allow us to normalize the prior and the likelihood,  
  # putting them both in a probability metric  
  group_by(n_trials) %>%  
  mutate(prior      = prior      / sum(prior),  
        likelihood = likelihood / sum(likelihood)) %>%  
  
  # plot!  
  ggplot(aes(x = p_water)) +  
    geom_line(aes(y = prior), linetype = 2) +  
    geom_line(aes(y = likelihood)) +  
    scale_x_continuous("proportion water", breaks = c(0, .5, 1)) +  
    scale_y_continuous("plausibility", breaks = NULL) +  
    theme(panel.grid = element_blank()) +  
    facet_wrap(~strip, scales = "free_y")
```



If it wasn't clear in the code, the dashed curves are normalized prior densities. The solid ones are normalized likelihoods. If you don't normalize (i.e., divide the density by the sum of the density), their respective heights don't match up with those in the text. Furthermore, it's the normalization that makes them directly comparable.

To learn more about `dplyr::group_by()` and its opposite `dplyr::ungroup()`, check out [R4DS, Chapter 5](#). To learn about `tidyverse::expand()`, go [here](#).

2.2.3 Evaluate.

It's worth repeating the **Rethinking: Deflationary statistics** box, here.

It may be that Bayesian inference is the best general purpose method of inference known. However, Bayesian inference is much less powerful than we'd like it to be. There is no approach to inference that provides universal guarantees. No branch of applied mathematics has unfettered access to reality, because math is not discovered, like the proton. Instead it is invented, like the shovel. (p. 32)

2.3 Components of the model

1. a likelihood function: “the number of ways each conjecture could produce an observation”
2. one or more parameters: “the accumulated number of ways each conjecture cold produce the entire data”
3. a prior: “the initial plausibility of each conjectured cause of the data”

2.3.1 Likelihood.

If you let the count of water be w and the number of tosses be n , then the binomial likelihood may be expressed as:

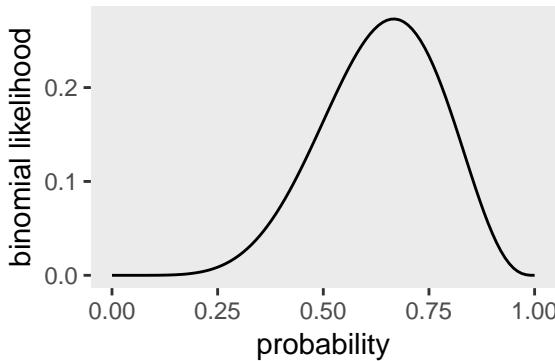
$$\Pr(w|n, p) = \frac{n!}{w!(n-w)!} p^w (1-p)^{n-w}$$

Given a probability of .5, the binomial likelihood of 6 out of 9 tosses coming out water is:

```
dbinom(x = 6, size = 9, prob = .5)
## [1] 0.1640625
```

McElreath suggested we change the values of `prob`. Let's do so over the parameter space.

```
tibble(prob = seq(from = 0, to = 1, by = .01)) %>%
  ggplot(aes(x = prob,
             y = dbinom(x = 6, size = 9, prob = prob))) +
  geom_line() +
  labs(x = "probability",
       y = "binomial likelihood") +
  theme(panel.grid = element_blank())
```



2.3.2 Parameters.

McElreath started off his **Rethinking: Datum or parameter?** box with:

It is typical to conceive of data and parameters as completely different kinds of entities. Data are measures and known; parameters are unknown and must be estimated from data. Usefully, in the Bayesian framework the distinction between a datum and a parameter is fuzzy. (p. 34)

For more in this topic, check out his lecture [Understanding Bayesian Statistics without Frequentist Language](#).

2.3.3 Prior.

So where do priors come from? They are engineering assumptions, chosen to help the machine learn. The flat prior in Figure 2.5 is very common, but it is hardly ever the best prior. You'll see later in the book that priors that gently nudge the machine usually improve inference. Such priors are sometimes called regularizing or weakly informative priors. (p. 35)

To learn more about “regularizing or weakly informative priors,” check out the [Prior Choice Recommendations](#) wiki from the Stan team.

2.3.3.1 Overthinking: Prior as a probability distribution

McElreath said that “for a uniform prior from a to b , the probability of any point in the interval is $1/(b - a)$ ” (p. 35). Let's try that out. To keep things simple, we'll hold a constant while varying the values for b .

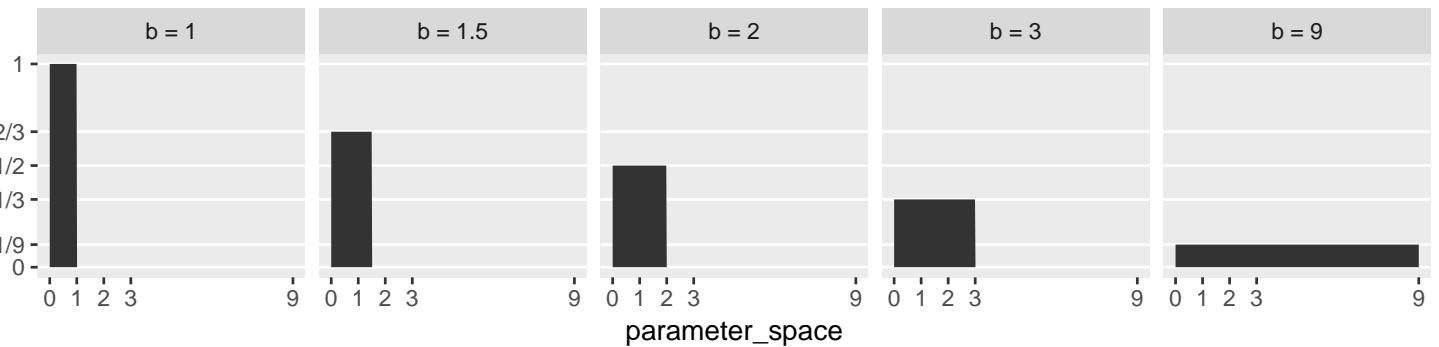
```
tibble(a = 0,
      b = c(1, 1.5, 2, 3, 9)) %>%
  mutate(prob = 1 / (b - a))
```

```
## # A tibble: 5 x 3
##       a     b   prob
##   <dbl> <dbl> <dbl>
## 1     0     1     1
## 2     0     1.5   0.667
## 3     0     2     0.5
## 4     0     3     0.333
## 5     0     9     0.111
```

I like to verify things with plots.

```
tibble(a = 0,
      b = c(1, 1.5, 2, 3, 9)) %>%
  expand(nesting(a, b), parameter_space = seq(from = 0, to = 9, length.out = 500)) %>%
  mutate(prob = dunif(parameter_space, a, b),
        b = str_c("b = ", b)) %>%

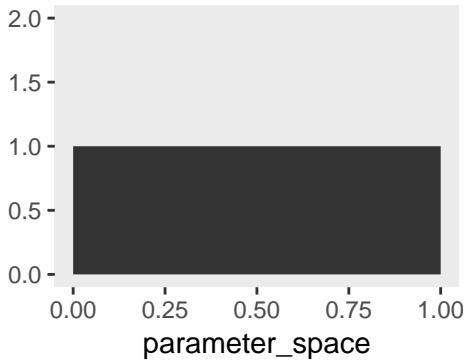
ggplot(aes(x = parameter_space, ymin = 0, ymax = prob)) +
  geom_ribbon() +
  scale_x_continuous(breaks = c(0, 1:3, 9)) +
  scale_y_continuous(breaks = c(0, 1/9, 1/3, 1/2, 2/3, 1),
                     labels = c("0", "1/9", "1/3", "1/2", "2/3", "1")) +
  theme(panel.grid.minor = element_blank(),
        panel.grid.major.x = element_blank()) +
  facet_wrap(~b, ncol = 5)
```



And as we'll learn much later in the project, the $\text{Uniform}(0, 1)$ distribution is special in that we can also express it as the beta distribution for which $\alpha = 1$ and $\beta = 1$. E.g.,

```
tibble(parameter_space = seq(from = 0, to = 1, length.out = 50)) %>%
  mutate(prob = dbeta(parameter_space, 1, 1)) %>%

ggplot(aes(x = parameter_space, ymin = 0, ymax = prob)) +
  geom_ribbon() +
  coord_cartesian(ylim = 0:2) +
  theme(panel.grid = element_blank())
```



2.3.4 Posterior.

If we continue to focus on the globe tossing example, the posterior probability a toss will be water may be expressed as:

$$\Pr(p|w) = \frac{\Pr(w|p)\Pr(p)}{\Pr(w)}$$

More generically and in words, this is:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Average Likelihood}}$$

2.4 Making the model go

Here's the data wrangling for Figure 2.6.

```
sequence_length <- 1e3

d <-
  tibble(probability = seq(from = 0, to = 1, length.out = sequence_length)) %>%
  expand(probability, row = c("flat", "stepped", "Laplace")) %>%
  arrange(row, probability) %>%
  mutate(prior = ifelse(row == "flat", 1,
                        ifelse(row == "stepped", rep(0:1, each = sequence_length / 2),
                               exp(-abs(probability - .5) / .25) / (2 * .25))),
         likelihood = dbinom(x = 6, size = 9, prob = probability)) %>%
  group_by(row) %>%
  mutate(posterior = prior * likelihood / sum(prior * likelihood)) %>%
  gather(key, value, -probability, -row) %>%
  ungroup() %>%
  mutate(key = factor(key, levels = c("prior", "likelihood", "posterior")),
         row = factor(row, levels = c("flat", "stepped", "Laplace")))
```

To learn more about `dplyr::arrange()`, check out [R4DS, Chapter 5.3](#).

In order to avoid unnecessary facet labels for the rows, it was easier to just make each column of the plot separately and then recombine them with `gridExtra::grid.arrange()`.

```
p1 <-
  d %>%
  filter(key == "prior") %>%
  ggplot(aes(x = probability, y = value)) +
  geom_line() +
  scale_x_continuous(NULL, breaks = c(0, .5, 1)) +
  scale_y_continuous(NULL, breaks = NULL) +
```

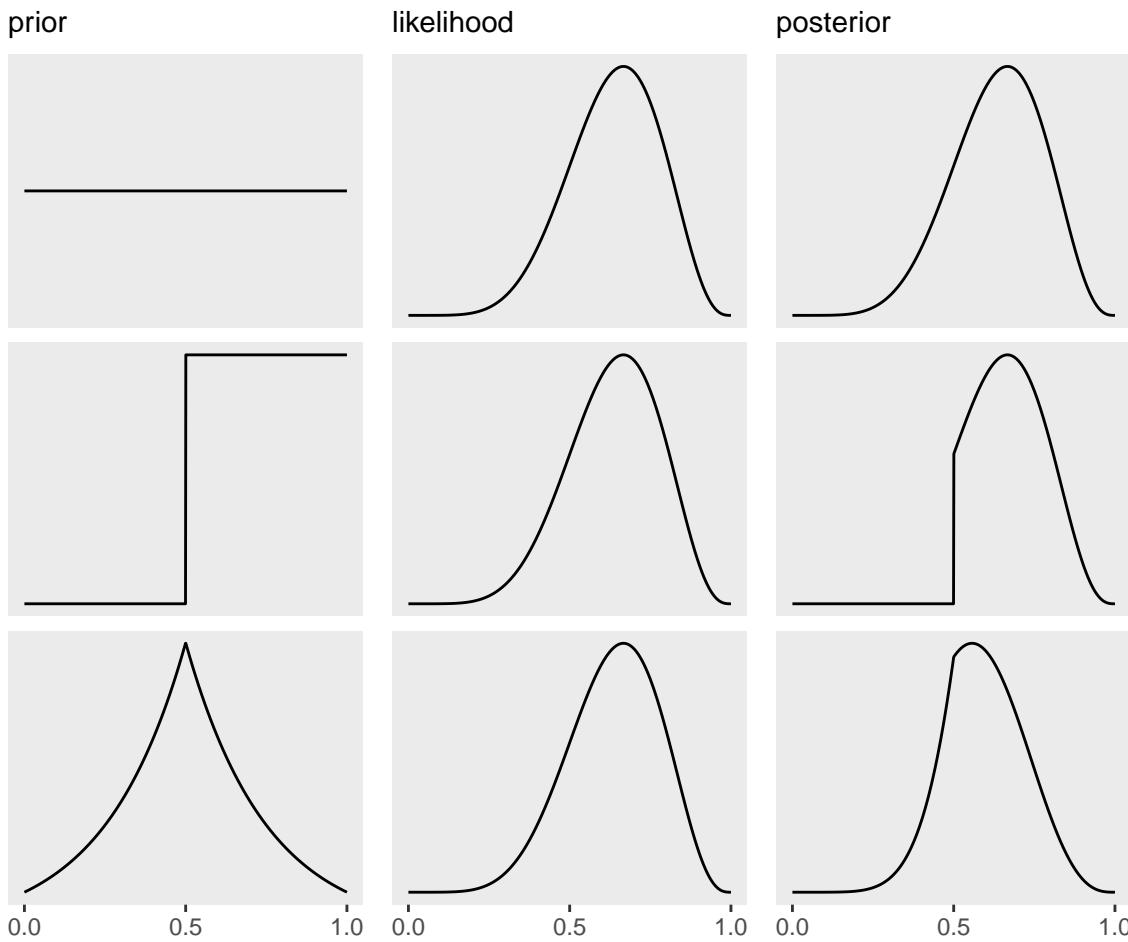
```
labs(subtitle = "prior") +
theme(panel.grid      = element_blank(),
  strip.background = element_blank(),
  strip.text       = element_blank()) +
facet_wrap(row ~ ., scales = "free_y", ncol = 1)

p2 <-
d %>%
filter(key == "likelihood") %>%
ggplot(aes(x = probability, y = value)) +
geom_line() +
scale_x_continuous(NULL, breaks = c(0, .5, 1)) +
scale_y_continuous(NULL, breaks = NULL) +
labs(subtitle = "likelihood") +
theme(panel.grid      = element_blank(),
  strip.background = element_blank(),
  strip.text       = element_blank()) +
facet_wrap(row ~ ., scales = "free_y", ncol = 1)

p3 <-
d %>%
filter(key == "posterior") %>%
ggplot(aes(x = probability, y = value)) +
geom_line() +
scale_x_continuous(NULL, breaks = c(0, .5, 1)) +
scale_y_continuous(NULL, breaks = NULL) +
labs(subtitle = "posterior") +
theme(panel.grid      = element_blank(),
  strip.background = element_blank(),
  strip.text       = element_blank()) +
facet_wrap(row ~ ., scales = "free_y", ncol = 1)

library(gridExtra)

grid.arrange(p1, p2, p3, ncol = 3)
```



I'm not sure if it's the same McElreath used in the text, but the formula I used for the triangle-shaped prior is the [Laplace distribution](#) with a location of .5 and a dispersion of .25.

Also, to learn all about `dplyr::filter()`, check out Baert's [Data Wrangling Part 3: Basic and more advanced ways to filter rows](#).

2.4.1 Grid approximation.

We just employed grid approximation over the last figure. In order to get nice smooth lines, we computed the posterior over 1000 evenly-spaced points on the probability space. Here we'll prepare for Figure 2.7 with 20.

```
(d <-  
  tibble(p_grid = seq(from = 0, to = 1, length.out = 20), # define grid  
         prior = 1) %>% # define prior  
  mutate(likelihood = dbinom(6, size = 9, prob = p_grid)) %>% # compute likelihood at each value in grid  
  mutate(unstd_posterior = likelihood * prior) %>% # compute product of likelihood and prior  
  mutate(posterior = unstd_posterior / sum(unstd_posterior)) # standardize the posterior, so it sums to 1  
)  
  
## # A tibble: 20 x 5  
##   p_grid prior likelihood unstd_posterior    posterior  
##     <dbl> <dbl>       <dbl>        <dbl>        <dbl>  
## 1 0      1 0          0          0  
## 2 0.0526 1 0.00000152 0.00000152 0.000000799  
## 3 0.105  1 0.00000819 0.00000819 0.00000431  
## 4 0.158  1 0.000777  0.000777  0.000409  
## 5 0.211  1 0.00360   0.00360   0.00189  
## 6 0.263  1 0.0112   0.0112   0.00587  
## 7 0.316  1 0.0267   0.0267   0.0140  
## 8 0.368  1 0.0529   0.0529   0.0279
```

```

## 9 0.421      1 0.0908      0.0908      0.0478
## 10 0.474     1 0.138       0.138       0.0728
## 11 0.526     1 0.190       0.190       0.0999
## 12 0.579     1 0.236       0.236       0.124
## 13 0.632     1 0.267       0.267       0.140
## 14 0.684     1 0.271       0.271       0.143
## 15 0.737     1 0.245       0.245       0.129
## 16 0.789     1 0.190       0.190       0.0999
## 17 0.842     1 0.118       0.118       0.0621
## 18 0.895     1 0.0503      0.0503      0.0265
## 19 0.947     1 0.00885     0.00885     0.00466
## 20 1          1 0           0           0

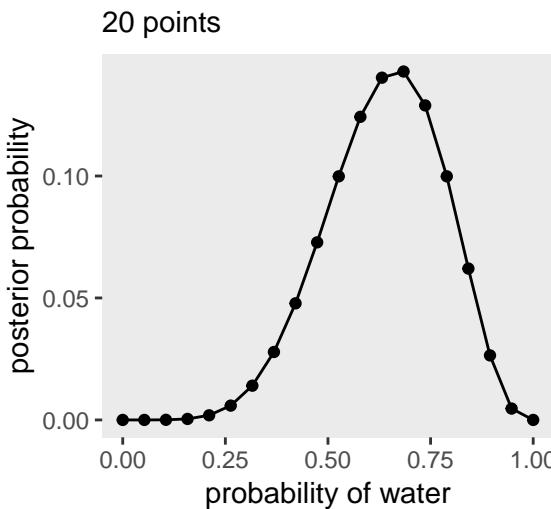
```

Here's the right panel of Figure 2.7.

```

d %>%
  ggplot(aes(x = p_grid, y = posterior)) +
  geom_point() +
  geom_line() +
  labs(subtitle = "20 points",
       x = "probability of water",
       y = "posterior probability") +
  theme(panel.grid = element_blank())

```



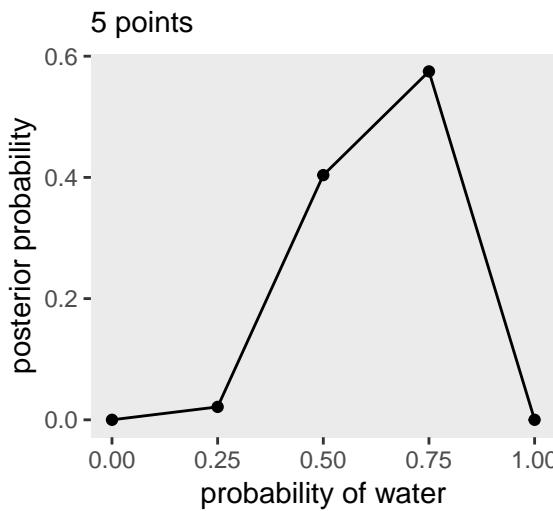
Here it is with just 5 points, the left hand panel of Figure 2.7.

```

tibble(p_grid            = seq(from = 0, to = 1, length.out = 5),
       prior             = 1) %>%
  mutate(likelihood        = dbinom(6, size = 9, prob = p_grid)) %>%
  mutate(unstd_posterior = likelihood * prior) %>%
  mutate(posterior        = unstd_posterior / sum(unstd_posterior)) %>%

  ggplot(aes(x = p_grid, y = posterior)) +
  geom_point() +
  geom_line() +
  labs(subtitle = "5 points",
       x = "probability of water",
       y = "posterior probability") +
  theme(panel.grid = element_blank())

```



2.4.2 Quadratic approximation.

Apply the quadratic approximation to the globe tossing data with `rethinking::map()`.

```
library(rethinking)

globe_qa <-
  rethinking::map(
    alist(
      w ~ dbinom(9, p), # binomial likelihood
      p ~ dunif(0, 1)   # uniform prior
    ),
    data = list(w = 6))

# display summary of quadratic approximation
precis(globe_qa)
```

	mean	sd	5.5%	94.5%
p	0.6666679	0.1571335	0.4155383	0.9177976

In preparation for Figure 2.8, here's the model with $n = 18$ and $n = 36$.

```
globe_qa_18 <-
  rethinking::map(
    alist(
      w ~ dbinom(9 * 2, p),
      p ~ dunif(0, 1)
    ),
    data = list(w = 6 * 2))

globe_qa_36 <-
  rethinking::map(
    alist(
      w ~ dbinom(9 * 4, p),
      p ~ dunif(0, 1)
    ),
    data = list(w = 6 * 4))

precis(globe_qa_18)

##           mean         sd      5.5%     94.5%
## p 0.6666665 0.1111104 0.4890906 0.8442423
```

```
precis(globe_qa_36)
```

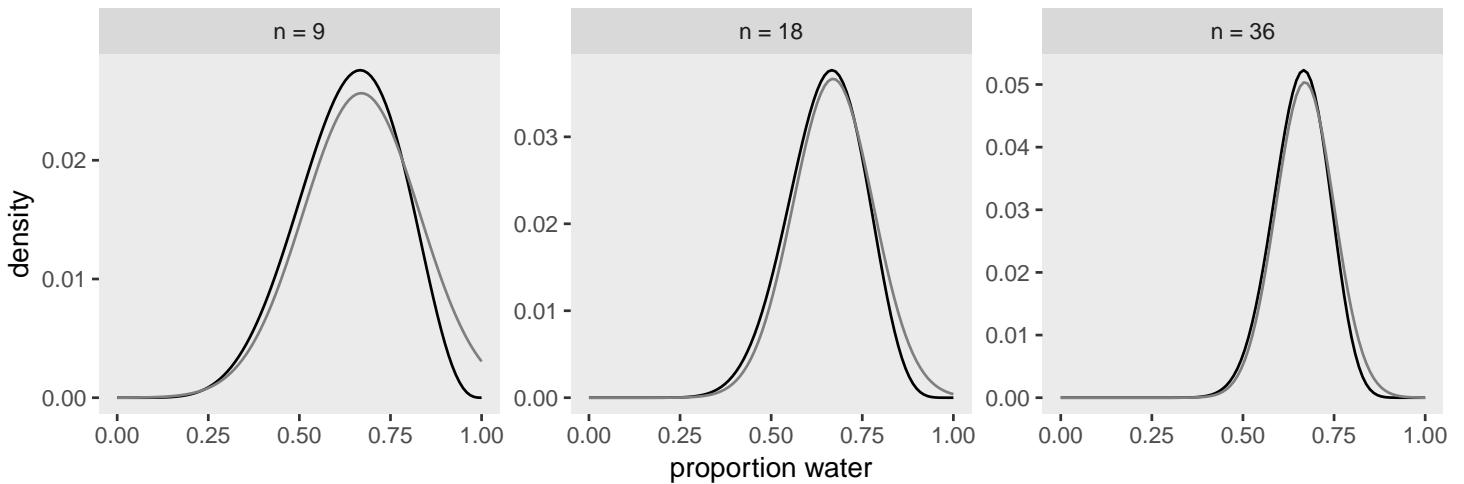
```
##      mean      sd    5.5%   94.5%
## p 0.6666691 0.0785666 0.5411045 0.7922337
```

Here's the legwork for Figure 2.8.

```
n_grid <- 100

tibble(p_grid
       prior
       w
       n
       m
       s
       mutate(likelihood
              = dbinom(w, size = n, prob = p_grid)) %>%
       mutate(unstd_grid_posterior = likelihood * prior,
              unstd_quad_posterior = dnorm(p_grid, m, s)) %>%
       group_by(w) %>%
       mutate(grid_posterior = unstd_grid_posterior / sum(unstd_grid_posterior),
              quad_posterior = unstd_quad_posterior / sum(unstd_quad_posterior),
              n = str_c("n = ", n)) %>%
       mutate(n = factor(n, levels = c("n = 9", "n = 18", "n = 36"))) %>%

ggplot(aes(x = p_grid)) +
  geom_line(aes(y = grid_posterior)) +
  geom_line(aes(y = quad_posterior),
            color = "grey50") +
  labs(x = "proportion water",
       y = "density") +
  theme(panel.grid = element_blank()) +
  facet_wrap(~n, scales = "free")
```



2.4.3 Markov chain Monte Carlo.

Since the main goal of this project is to highlight brms, we may as fit a model. This seems like an appropriately named subsection to do so. First we'll have to load the package.

```
library(brms)
```

Here we'll re-fit the last model from above wherein $w = 24$ and $n = 36$.

```
globe_qa_brms <-
  brm(data = list(w = 24),
       family = binomial(link = "identity"),
       w | trials(36) ~ 1,
       prior(beta(1, 1), class = Intercept),
       iter = 4000, warmup = 1000,
       control = list(adapt_delta = .9),
       seed = 4)
```

The model output looks like so.

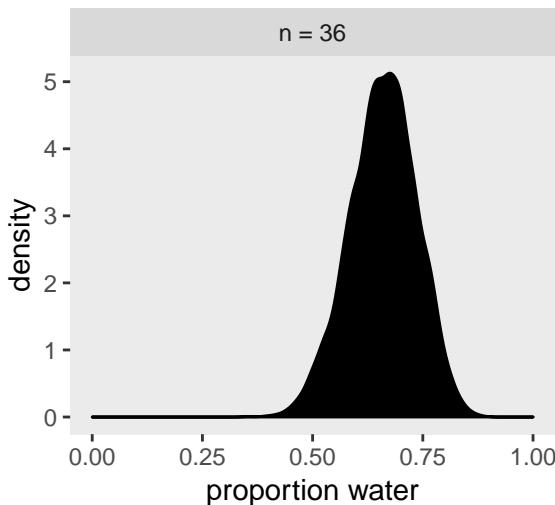
```
print(globe_qa_brms)

## Family: binomial
## Links: mu = identity
## Formula: w | trials(36) ~ 1
## Data: list(w = 24) (Number of observations: 1)
## Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 12000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     0.66      0.08     0.50     0.80        3579  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

There's a lot going on in that output, which we'll start to clarify in Chapter 4. For now, focus on the 'Intercept' line. As we'll also learn in Chapter 4, the intercept of a regression model with no predictors is the same as its mean. In the special case of a model using the binomial likelihood, the mean is the probability of a 1 in a given trial, θ .

Let's plot the results of our model and compare them with those from `rethinking::map()`, above.

```
posterior_samples(globe_qa_brms) %>%
  mutate(n = "n = 36") %>%
  ggplot(aes(x = b_Intercept)) +
  geom_density(fill = "black") +
  labs(x = "proportion water") +
  xlim(0, 1) +
  theme(panel.grid = element_blank()) +
  facet_wrap(~n)
```



If you're still confused. Cool. This is just a preview. We'll start walking through fitting models in brms in Chapter 4 and we'll learn a lot about regression with the binomial likelihood in Chapter 10.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods  base
##
## other attached packages:
## [1] brms_2.8.8           Rcpp_1.0.1          rethinking_1.80    rstan_2.18.2
## [5] StanHeaders_2.18.0-1 gridExtra_2.3  forcats_0.3.0     stringr_1.4.0
## [9] dplyr_0.8.0.1        purrr_0.2.5       readr_1.1.1       tidyverse_1.2.1
## [13] tibble_2.1.1         ggplot2_3.1.1     tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
## [1] nlme_3.1-137      matrixStats_0.54.0 xts_0.10-2       lubridate_1.7.4
## [5] threejs_0.3.1     httr_1.3.1        rprojroot_1.3-2 tools_3.5.1
## [9] backports_1.1.4   DT_0.4            utf8_1.1.4       R6_2.3.0
## [13] lazyeval_0.2.2    colorspace_1.3-2  withr_2.1.2       tidyselect_0.2.5
## [17] prettyunits_1.0.2 processx_3.2.1    Brobdingnag_1.2-6 compiler_3.5.1
## [21] cli_1.0.1         rvest_0.3.2       shinyjs_1.0       xml2_1.2.0
## [25] colourpicker_1.0 labeling_0.3      bookdown_0.9      scales_1.0.0
## [29] dygraphs_1.1.1.5 mvtnorm_1.0-10   ggridges_0.5.0   callr_3.1.0
## [33] digest_0.6.18    rmarkdown_1.10   base64enc_0.1-3  pkgconfig_2.0.2
## [37] htmltools_0.3.6   htmlwidgets_1.2   rlang_0.3.4       readxl_1.1.0
## [41] rstudioapi_0.7    shiny_1.1.0       generics_0.0.2    zoo_1.8-2
## [45] jsonlite_1.5      gtools_3.8.1     crosstalk_1.0.0   inline_0.3.15
## [49] magrittr_1.5       loo_2.1.0        bayesplot_1.6.0  Matrix_1.2-14
## [53] munsell_0.5.0     fansi_0.4.0      abind_1.4-5      stringi_1.4.3
## [57] yaml_2.1.19       MASS_7.3-50      pkgbuild_1.0.2   plyr_1.8.4
## [61] grid_3.5.1        promises_1.0.1   crayon_1.3.4     miniUI_0.1.1.1
## [65] lattice_0.20-35  haven_1.1.2     hms_0.4.2        knitr_1.20
## [69] ps_1.2.1          pillar_1.3.1     igraph_1.2.1     markdown_0.8
## [73] shinystan_2.5.0   reshape2_1.4.3   stats4_3.5.1    rstantools_1.5.1
## [77] glue_1.3.1.9000  evaluate_0.10.1  modelr_0.1.2    httpuv_1.4.4.2
## [81] cellranger_1.1.0  gtable_0.3.0     assertthat_0.2.0 xfun_0.3
## [85] mime_0.5          xtable_1.8-2     broom_0.5.1      coda_0.19-2
## [89] later_0.7.3      rsconnect_0.8.8  shinythemes_1.1.1 bridgesampling_0.6-0
```

Chapter 3

Sampling the Imaginary

If you would like to know the probability someone is a vampire given they test positive to the blood-based vampire test, you compute

$$\text{Pr}(\text{vampire}|\text{positive}) = \frac{\text{Pr}(\text{positive}|\text{vampire}) \text{ Pr}(\text{vampire})}{\text{Pr}(\text{positive})}$$

We'll do so within a tibble.

```
library(tidyverse)

tibble(pr_positive_vampire = .95,
      pr_positive_mortal   = .01,
      pr_vampire           = .001) %>%
  mutate(pr_positive       = pr_positive_vampire * pr_vampire + pr_positive_mortal * (1 - pr_vampire)) %>%
  mutate(pr_vampire_positive = pr_positive_vampire * pr_vampire / pr_positive) %>%
  glimpse()
```

```
## Observations: 1
## Variables: 5
## $ pr_positive_vampire <dbl> 0.95
## $ pr_positive_mortal <dbl> 0.01
## $ pr_vampire          <dbl> 0.001
## $ pr_positive         <dbl> 0.01094
## $ pr_vampire_positive <dbl> 0.08683729
```

Here's the other way of tackling the vampire problem, this time useing the frequency format.

```
tibble(pr_vampire        = 100 / 100000,
      pr_positive_vampire = 95 / 100,
      pr_positive_mortal = 99 / 99900) %>%
  mutate(pr_positive       = 95 + 999) %>%
  mutate(pr_vampire_positive = pr_positive_vampire * 100 / pr_positive) %>%
  glimpse()
```

```
## Observations: 1
## Variables: 5
## $ pr_vampire          <dbl> 0.001
## $ pr_positive_vampire <dbl> 0.95
## $ pr_positive_mortal  <dbl> 0.000990991
## $ pr_positive          <dbl> 1094
## $ pr_vampire_positive <dbl> 0.08683729
```

3.1 Sampling from a grid-like approximate posterior

Here we use grid approximation, again, to generate samples.

```
# how many grid points would you like?
n <- 1001
n_success <- 6
n_trials <- 9

(
  d <-
  tibble(p_grid      = seq(from = 0, to = 1, length.out = n),
         # note we're still using a flat uniform prior
         prior       = 1) %>%
  mutate(likelihood = dbinom(n_success, size = n_trials, prob = p_grid)) %>%
  mutate(posterior  = (likelihood * prior) / sum(likelihood * prior))
)

## # A tibble: 1,001 x 4
##   p_grid prior likelihood posterior
##   <dbl> <dbl>     <dbl>     <dbl>
## 1 0        1     0.          0.
## 2 0.001    1     8.37e-17  8.37e-19
## 3 0.002    1     5.34e-15  5.34e-17
## 4 0.003    1     6.07e-14  6.07e-16
## 5 0.004    1     3.40e-13  3.40e-15
## 6 0.005    1     1.29e-12  1.29e-14
## 7 0.006    1     3.85e-12  3.85e-14
## 8 0.007    1     9.68e-12  9.68e-14
## 9 0.008    1     2.15e-11  2.15e-13
## 10 0.009   1     4.34e-11  4.34e-13
## # ... with 991 more rows
```

Now we'll use the `dplyr::sample_n()` function to sample rows from `d`, saving them as `sample`.

```
# how many samples would you like?
n_samples <- 1e4

# make it reproducible
set.seed(3)

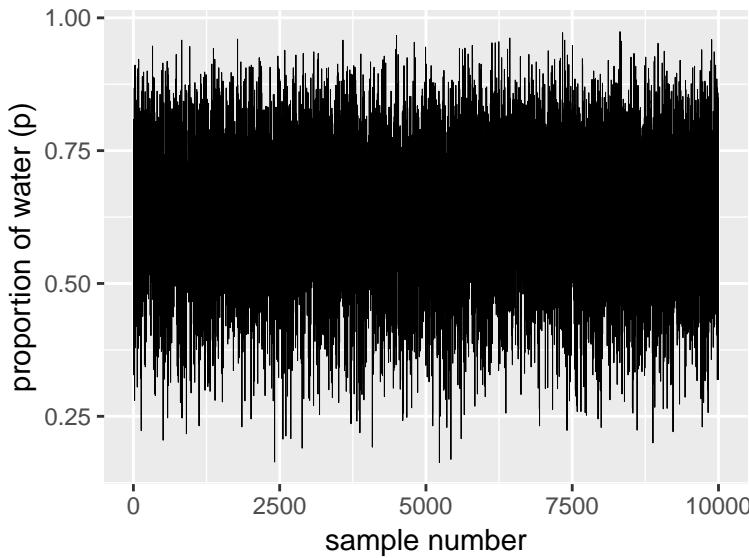
samples <-
d %>%
  sample_n(size = n_samples, weight = posterior, replace = T)

glimpse(samples)

## Observations: 10,000
## Variables: 4
## $ p_grid      <dbl> 0.734, 0.808, 0.385, 0.328, 0.510, 0.511, 0.756, 0.674, 0.578, 0.520, 0.512, ...
## $ prior       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ likelihood <dbl> 0.24722965, 0.16544404, 0.06363104, 0.03174170, 0.17389560, 0.17487642, 0.22...
## $ posterior  <dbl> 0.0024722965, 0.0016544404, 0.0006363104, 0.0003174170, 0.0017389560, 0.0017...
```

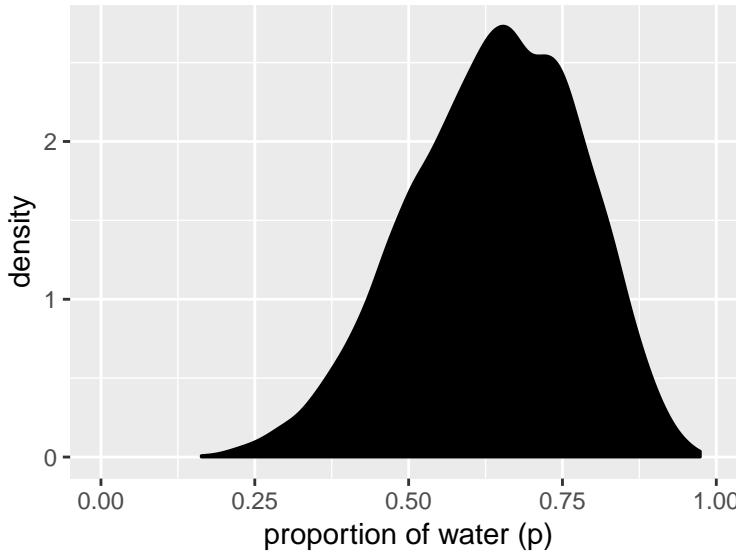
We'll plot the zigzagging left panel of Figure 3.1 with `geom_line()`. But before we do, we'll need to add a variable numbering the samples.

```
samples %>%
  mutate(sample_number = 1:n()) %>%
  ggplot(aes(x = sample_number, y = p_grid)) +
  geom_line(size = 1/10) +
  labs(x = "sample number",
       y = "proportion of water (p)")
```



We'll make the density in the right panel with `geom_density()`.

```
samples %>%
  ggplot(aes(x = p_grid)) +
  geom_density(fill = "black") +
  coord_cartesian(xlim = 0:1) +
  xlab("proportion of water (p)")
```



3.2 Sampling to summarize

"Once your model produces a posterior distribution, the model's work is done. But your work has just begun. It is necessary to summarize and interpret the posterior distribution. Exactly now it is summarized depends upon your purpose" (p. 53).

3.2.1 Intervals of defined boundaries.

To get the proportion of water less than some value of `p_grid` within the tidyverse, you'd first `filter()` by that value and then take the `sum()` within `summarise()`.

```
d %>%
  filter(p_grid < .5) %>%
  summarise(sum = sum(posterior))
```

```
## # A tibble: 1 x 1
##       sum
##     <dbl>
## 1 0.171
```

To learn more about `dplyr::summarise()` and related functions, check out Baert's *Data Wrangling Part 4: Summarizing and slicing your data* and Chapter 5.6 of *R4DS*.

If what you want is a frequency based on filtering by `samples`, then you might use `n()` within `summarise()`.

```
samples %>%
  filter(p_grid < .5) %>%
  summarise(sum = n() / n_samples)
```

```
## # A tibble: 1 x 1
##       sum
##     <dbl>
## 1 0.169
```

You can use `&` within `filter()`, too.

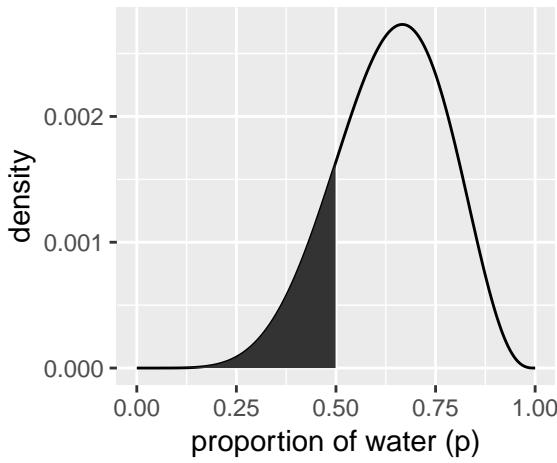
```
samples %>%
  filter(p_grid > .5 & p_grid < .75) %>%
  summarise(sum = n() / n_samples)
```

```
## # A tibble: 1 x 1
##       sum
##     <dbl>
## 1 0.596
```

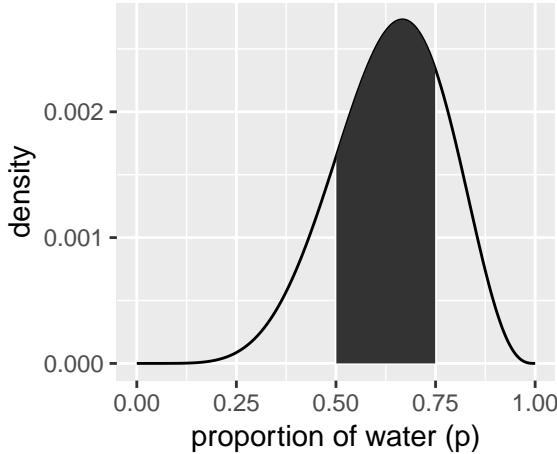
3.2.2 Intervals of defined mass.

We'll create the upper two panels for Figure 3.2 with `geom_line()`, `geom_ribbon()`, and a some careful filtering.

```
# upper left panel
d %>%
  ggplot(aes(x = p_grid)) +
  geom_line(aes(y = posterior)) +
  geom_ribbon(data = d %>% filter(p_grid < .5),
              aes(ymax = 0, ymin = posterior)) +
  labs(x = "proportion of water (p)",
       y = "density")
```



```
# upper right panel
d %>%
  ggplot(aes(x = p_grid)) +
  geom_line(aes(y = posterior)) +
  # note this next line is the only difference in code from the last plot
  geom_ribbon(data = d %>% filter(p_grid < .75 & p_grid > .5),
              aes(ymax = 0, ymin = posterior)) +
  labs(x = "proportion of water (p)",
       y = "density")
```



We'll come back for the lower two panels in a bit.

Since we've saved our `p_grid` samples within the well-named `samples` tibble, we'll have to index with \$ within `quantile`.

```
(q_80 <- quantile(samples$p_grid, prob = .8))
```

```
##   80%
## 0.763
```

That value will come in handy for the lower left panel of Figure 3.2, so we saved it. But anyways, we could `select()` the `samples` vector, extract it from the tibble with `pull()`, and then pump it into `quantile()`:

```
samples %>%
  select(p_grid) %>%
  pull() %>%
  quantile(prob = .8)
```

```
##   80%
## 0.763
```

And we might also use `quantile()` within `summarise()`.

```
samples %>%
  summarise(`80th percentile` = quantile(p_grid, p = .8))

## # A tibble: 1 x 1
##   `80th percentile`
##             <dbl>
## 1           0.763
```

Here's the `summarise()` approach with two probabilities:

```
samples %>%
  summarise(`10th percentile` = quantile(p_grid, p = .1),
            `90th percentile` = quantile(p_grid, p = .9))

## # A tibble: 1 x 2
##   `10th percentile` `90th percentile`
##             <dbl>             <dbl>
## 1           0.451           0.814
```

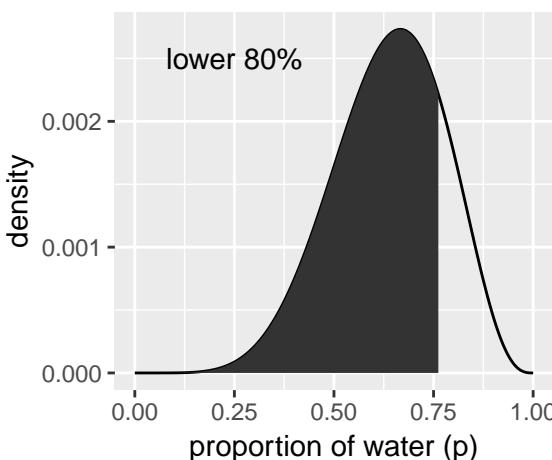
The `tidyverse` approach is nice in that that family of functions typically returns a data frame. But sometimes you just want your values in a numeric vector for the sake of quick indexing. In that case, base R `quantile()` shines.

```
(q_10_and_90 <- quantile(samples$p_grid, prob = c(.1, .9)))

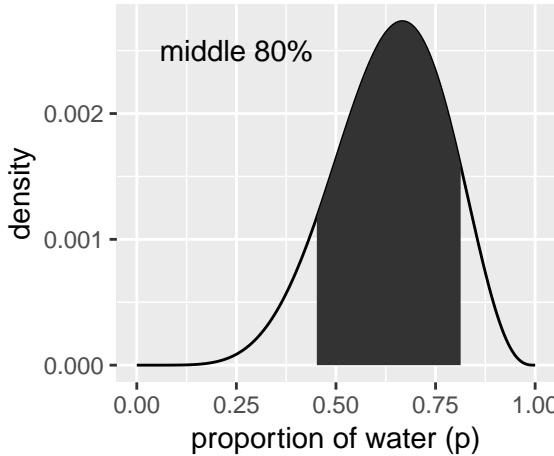
##    10%    90%
## 0.451 0.814
```

Now we have our cutoff values saved as `q_80` and `q_10_and_90`, we're ready to make the bottom panels of Figure 3.2.

```
# lower left panel
d %>%
  ggplot(aes(x = p_grid)) +
  geom_line(aes(y = posterior)) +
  geom_ribbon(data = d %>% filter(p_grid < q_80),
              aes(ymax = 0, ymin = posterior)) +
  annotate(geom = "text",
           x = .25, y = .0025,
           label = "lower 80%") +
  labs(x = "proportion of water (p)",
       y = "density")
```



```
# lower right panel
d %>%
  ggplot(aes(x = p_grid)) +
  geom_line(aes(y = posterior)) +
  geom_ribbon(data = d %>% filter(p_grid > q_10_and_90[1] & p_grid < q_10_and_90[2]),
              aes(ymax = 0, ymin = posterior)) +
  annotate(geom = "text",
          x = .25, y = .0025,
          label = "middle 80%") +
  labs(x = "proportion of water (p)",
       y = "density")
```



We've already defined `p_grid` and `prior` within `d`, above. Here we'll reuse them and update the rest of the columns.

```
# here we update the `dbinom()` parameters
n_success <- 3
n_trials  <- 3

# update `d`
d <-
d %>%
  mutate(likelihood = dbinom(n_success, size = n_trials, prob = p_grid)) %>%
  mutate(posterior  = (likelihood * prior) / sum(posterior))
```

```
# make the next part reproducible
set.seed(3)
```

```
# here's our new samples tibble
(
  samples <-
  d %>%
    sample_n(size = n_samples, weight = posterior, replace = T)
)
```

```
## # A tibble: 10,000 x 4
##   p_grid prior likelihood posterior
##   <dbl> <dbl>     <dbl>      <dbl>
## 1 0.944  1        0.841      0.841
## 2 0.808  1        0.528      0.528
## 3 0.872  1        0.663      0.663
## 4 0.328  1        0.0353     0.0353
## 5 0.602  1        0.218      0.218
## 6 0.809  1        0.529      0.529
## 7 0.959  1        0.882      0.882
```

```
## 8 0.902 1 0.734 0.734
## 9 0.578 1 0.193 0.193
## 10 0.631 1 0.251 0.251
## # ... with 9,990 more rows
```

The `rethinking::PI()` function works like a nice shorthand for `quantile()`.

```
quantile(samples$p_grid, prob = c(.25, .75))
```

```
## 25% 75%
## 0.703 0.934
```

```
rethinking::PI(samples$p_grid, prob = .5)
```

```
## 25% 75%
## 0.703 0.934
```

Now's a good time to introduce Matthew Kay's [tidybayes package](#), which offers an [array of convenience functions](#) for Bayesian models of the type we'll be working with in this project.

```
library(tidybayes)
median_qi(samples$p_grid, .width = .5)
```

```
## y ymin ymax .width .point .interval
## 1 0.841 0.703 0.934 0.5 median qi
```

The tidybayes package offers a [family of functions](#) that make it easy to summarize a distribution with a measure of central tendency accompanied by intervals. With `median_qi()`, we asked for the median and quantile-based intervals—just like we've been doing with `quantile()`. Note how the `.width` argument within `median_qi()` worked the same way the `prob` argument did within `rethinking::PI()`. With `.width = .5`, we indicated we wanted a quantile-based 50% interval, which was returned in the `ymin` and `ymax` columns. The tidybayes framework makes it easy to request multiple types of intervals. E.g., here we'll request 50%, 80%, and 99% intervals.

```
median_qi(samples$p_grid, .width = c(.5, .8, .99))
```

```
## y ymin ymax .width .point .interval
## 1 0.841 0.703000 0.934 0.50 median qi
## 2 0.841 0.559000 0.975 0.80 median qi
## 3 0.841 0.265995 0.999 0.99 median qi
```

The `.width` column in the output indexed which line presented which interval.

Now let's use the `rethinking::HPDI()` function to return 50% highest posterior density intervals (HPDIs).

```
rethinking::HPDI(samples$p_grid, prob = .5)
```

```
## |0.5 0.5|
## 0.840 0.999
```

The reason I introduce tidybayes now is that the functions of the brms package only support percentile-based intervals of the type we computed with `quantile()` and `median_qi()`. But tidybayes also supports HPDIs.

```
mode_hdi(samples$p_grid, .width = .5)
```

```
##           y ymin  ymax .width .point .interval
## 1 0.9562364 0.84 0.999    0.5   mode      hdi
```

This time we used the mode as the measure of central tendency. With this family of tidybayes functions, you specify the measure of central tendency in the prefix (i.e., `mean`, `median`, or `mode`) and then the type of interval you'd like (i.e., `qi` or `hdi`).

If all you want are the intervals without the measure of central tendency or all that other technical information, tidybayes also offers the handy `qi()` and `hdi()` functions.

```
qi(samples$p_grid, .width = .5)
```

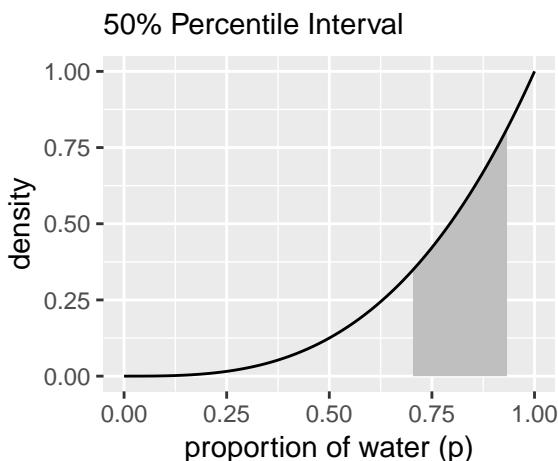
```
##      [,1]  [,2]
## [1,] 0.703 0.934
```

```
hdi(samples$p_grid, .width = .5)
```

```
##      [,1]  [,2]
## [1,] 0.84  0.999
```

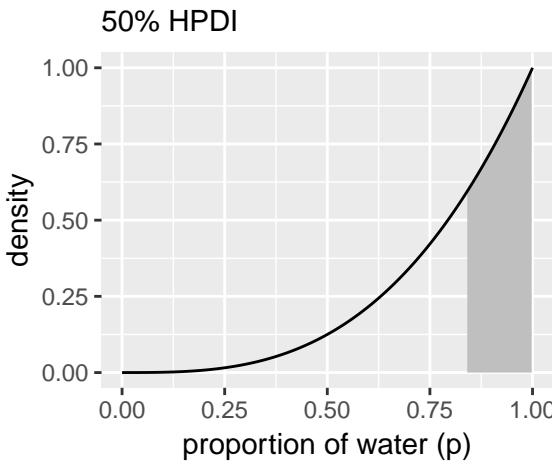
These are nice in that they yield simple numeric vectors, making them particularly useful to use as references within `ggplot2`. Now we have that skill, we can use it to make Figure 3.3.

```
# lower left panel
d %>%
  ggplot(aes(x = p_grid)) +
  # check out our sweet `qi()` ` indexing
  geom_ribbon(data = d %>% filter(p_grid > qi(samples$p_grid, .width = .5)[1] &
                                     p_grid < qi(samples$p_grid, .width = .5)[2]),
              aes(ymin = 0, ymax = posterior),
              fill = "grey75") +
  geom_line(aes(y = posterior)) +
  labs(subtitle = "50% Percentile Interval",
       x = "proportion of water (p)",
       y = "density")
```



```
# lower right panel
d %>%
  ggplot(aes(x = p_grid)) +
  geom_ribbon(data = d %>% filter(p_grid > hdi(samples$p_grid, .width = .5)[1] &
                                     p_grid < hdi(samples$p_grid, .width = .5)[2]),
              aes(ymin = 0, ymax = posterior),
```

```
fill = "grey75") +
geom_line(aes(y = posterior)) +
labs(subtitle = "50% HPDI",
x = "proportion of water (p)",
y = "density")
```



3.2.3 Point estimates.

We've been calling point estimates measures of central tendency. If we `arrange()` our `d` tibble in descending order by `posterior`, we'll see the corresponding `p_grid` value for its MAP estimate.

```
d %>%
arrange(desc(posterior))
```

```
## # A tibble: 1,001 x 4
##   p_grid prior likelihood posterior
##   <dbl> <dbl>      <dbl>      <dbl>
## 1 1     1         1         1
## 2 0.999 1         0.997     0.997
## 3 0.998 1         0.994     0.994
## 4 0.997 1         0.991     0.991
## 5 0.996 1         0.988     0.988
## 6 0.995 1         0.985     0.985
## 7 0.994 1         0.982     0.982
## 8 0.993 1         0.979     0.979
## 9 0.992 1         0.976     0.976
## 10 0.991 1         0.973     0.973
## # ... with 991 more rows
```

To emphasize it, we can use `slice()` to select the top row.

```
d %>%
arrange(desc(posterior)) %>%
slice(1)
```

```
## # A tibble: 1 x 4
##   p_grid prior likelihood posterior
##   <dbl> <dbl>      <dbl>      <dbl>
## 1 1     1         1         1
```

Or we could use the handy `dplyr::top_n()` function.

```
d %>%
  select(posterior) %>%
  top_n(n = 1)

## Selecting by posterior
```

```
## # A tibble: 1 x 1
##   posterior
##       <dbl>
## 1         1
```

We can get the mode with `mode_hdi()` or `mode_qi()`.

```
samples %>% mode_hdi(p_grid)
```

```
## # A tibble: 1 x 6
##   p_grid .lower .upper .width .point .interval
##       <dbl>  <dbl>  <dbl>  <dbl>  <chr>  <chr>
## 1  0.956   0.468     1    0.95 mode   hdi
```

```
samples %>% mode_qi(p_grid)
```

```
## # A tibble: 1 x 6
##   p_grid .lower .upper .width .point .interval
##       <dbl>  <dbl>  <dbl>  <dbl>  <chr>  <chr>
## 1  0.956   0.4    0.994   0.95 mode   qi
```

But if all you want is the mode itself, you can just use `tidybayes::Mode()`.

```
Mode(samples$p_grid)
```

```
## [1] 0.9562364
```

But medians and means are typical, too.

```
samples %>%
  summarise(mean = mean(p_grid),
            median = median(p_grid))
```

```
## # A tibble: 1 x 2
##   mean median
##     <dbl>  <dbl>
## 1  0.799  0.841
```

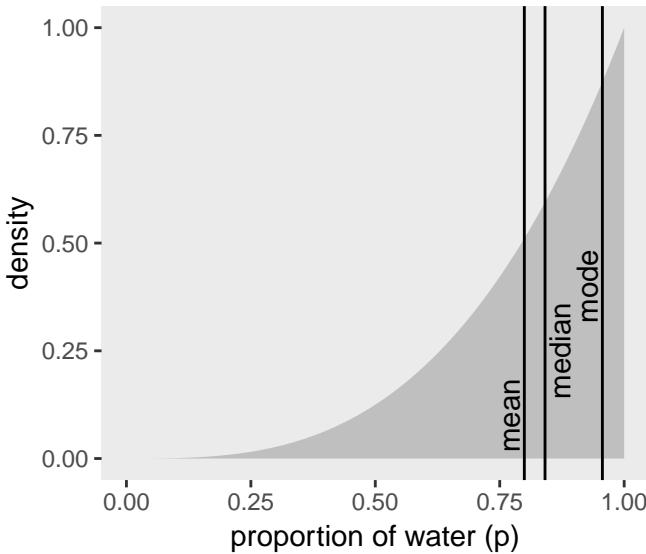
We can inspect the three types of point estimate in the left panel of Figure 3.4. First we'll bundle the three point estimates together in a tibble.

```
( point_estimates <-
  bind_rows(
    samples %>% mean_qi(p_grid),
    samples %>% median_qi(p_grid),
    samples %>% mode_qi(p_grid)
  ) %>%
  select(p_grid, .point) %>%
  # these last two columns will help us annotate
  mutate(x = p_grid + c(-.03, .03, -.03),
        y = c(.1, .25, .4))
)
```

```
## # A tibble: 3 x 4
##   p_grid .point      x      y
##   <dbl> <chr> <dbl> <dbl>
## 1 0.799 mean  0.769  0.1
## 2 0.841 median 0.871  0.25
## 3 0.956 mode   0.926  0.4
```

The plot:

```
d %>%
  ggplot(aes(x = p_grid)) +
  geom_ribbon(aes(ymin = 0, ymax = posterior),
              fill = "grey75") +
  geom_vline(xintercept = point_estimates$p_grid) +
  geom_text(data = point_estimates,
            aes(x = x, y = y, label = .point),
            angle = 90) +
  labs(x = "proportion of water (p)",
       y = "density") +
  theme(panel.grid = element_blank())
```



As it turns out “*different loss functions imply different point estimates*” (p. 59, *emphasis* in the original).

Let p be the proportion of the Earth covered by water and d be our guess. If McElreath pays us \$100 if we guess exactly right but subtracts money from the prize proportional to how far off we are, then our loss is proportional to $p - d$. If we decide $d = .5$, then our expected loss will be:

```
d %>%
  mutate(loss = posterior * abs(0.5 - p_grid)) %>%
  summarise(`expected loss` = sum(loss))
```

```
## # A tibble: 1 x 1
##   `expected loss`
##   <dbl>
## 1 78.4
```

What McElreath did with `sapply()`, we’ll do with `purrr::map()`. If you haven’t used it, `map()` is part of a family of similarly-named functions (e.g., `map2()`) from the [purrr package](#), which is itself part of the [tidyverse](#). The `map()` family is the tidyverse alternative to the family of `apply()` functions from the base R framework. You can learn more about how to use the `map()` family [here](#) or [here](#) or [here](#).

```

make_loss <- function(our_d){
  d %>%
    mutate(loss = posterior * abs(our_d - p_grid)) %>%
    summarise(weighted_average_loss = sum(loss))
}

(
  l <-
  d %>%
    select(p_grid) %>%
    rename(decision = p_grid) %>%
    mutate(weighted_average_loss = purrr::map(decision, make_loss)) %>%
    unnest()
)
## # A tibble: 1,001 x 2
##   decision weighted_average_loss
##       <dbl>              <dbl>
## 1 0                  201.
## 2 0.001             200.
## 3 0.002             200.
## 4 0.003             200.
## 5 0.004             199.
## 6 0.005             199.
## 7 0.006             199.
## 8 0.007             199.
## 9 0.008             198.
## 10 0.009            198.
## # ... with 991 more rows

```

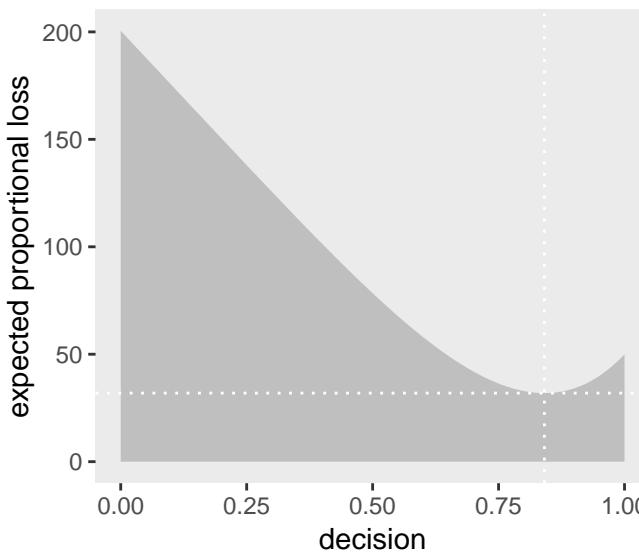
Now we're ready for the right panel of Figure 3.4.

```

# this will help us find the x and y coordinates for the minimum value
min_loss <-
  l %>%
  filter(weighted_average_loss == min(weighted_average_loss)) %>%
  as.numeric()

# the plot
l %>%
  ggplot(aes(x = decision)) +
  geom_ribbon(aes(ymin = 0, ymax = weighted_average_loss),
              fill = "grey75") +
  geom_vline(xintercept = min_loss[1], color = "white", linetype = 3) +
  geom_hline(yintercept = min_loss[2], color = "white", linetype = 3) +
  ylab("expected proportional loss") +
  theme(panel.grid = element_blank())

```



We saved the exact minimum value as `min_loss[1]`, which is 0.841. Within sampling error, this is the posterior median as depicted by our `samples`.

```
samples %>%
  summarise(posterior_median = median(p_grid))

## # A tibble: 1 x 1
##   posterior_median
##             <dbl>
## 1             0.841
```

The quadratic loss $(d - p)^2$ suggests we should use the mean instead. Let's investigate.

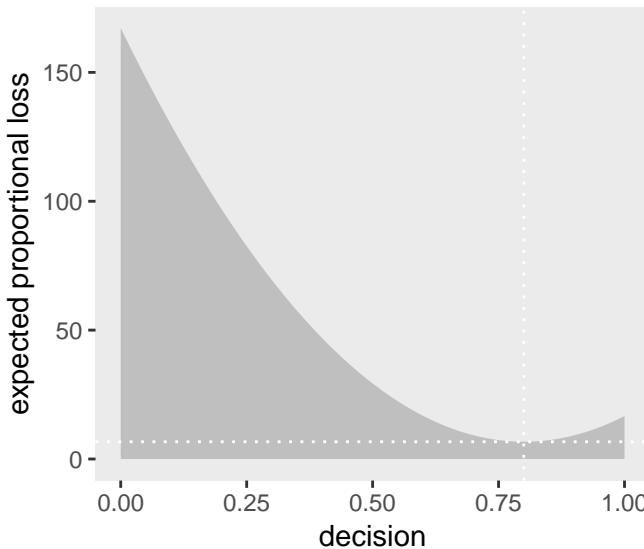
```
# amend our loss function
make_loss <- function(our_d){
  d %>%
    mutate(loss = posterior * (our_d - p_grid)^2) %>%
    summarise(weighted_average_loss = sum(loss))
}

# remake our `l` data
l <-
  d %>%
  select(p_grid) %>%
  rename(decision = p_grid) %>%
  mutate(weighted_average_loss = purrr::map(decision, make_loss)) %>%
  unnest()

# update to the new minimum loss coordinates
min_loss <-
  l %>%
  filter(weighted_average_loss == min(weighted_average_loss)) %>%
  as.numeric()

# update the plot
l %>%
  ggplot(aes(x = decision)) +
  geom_ribbon(aes(ymin = 0, ymax = weighted_average_loss),
              fill = "grey75") +
  geom_vline(xintercept = min_loss[1], color = "white", linetype = 3) +
  geom_hline(yintercept = min_loss[2], color = "white", linetype = 3) +
```

```
ylab("expected proportional loss") +
theme(panel.grid = element_blank())
```



Based on quadratic loss $(d - p)^2$, the exact minimum value is 0.8. Within sampling error, this is the posterior mean of our samples.

```
samples %>%
  summarise(posterior_mean = mean(p_grid))
```

```
## # A tibble: 1 x 1
##   posterior_mean
##       <dbl>
## 1         0.799
```

3.3 Sampling to simulate prediction

McElreath's four good reasons for posterior simulation were:

1. Model checking
2. Software validation
3. Research design
4. Forecasting

3.3.1 Dummy data.

Dummy data for the globe tossing model arise from the binomial likelihood. If you let w be a count of water and n be the number of tosses, the binomial likelihood is

$$\Pr(w|n, p) = \frac{n!}{w!(n-w)!} p^w (1-p)^{n-w}$$

Letting $n = 2$, $p(w) = .7$, and $w_{\text{observed}} = 0$ through 2, the densities are:

```
tibble(n      = 2,
       probability = .7,
       w          = 0:2) %>%
  mutate(density = dbinom(w, size = n, prob = probability))
```

```
## # A tibble: 3 x 4
##   n probability     w density
##   <dbl>      <dbl> <int>    <dbl>
## 1 2        0.7     0    0.09
## 2 2        0.7     1    0.42
## 3 2        0.7     2    0.490
```

If we're going to simulate, we should probably `set our seed`. Doing so makes the results reproducible.

```
set.seed(3)
rbinom(1, size = 2, prob = .7)

## [1] 2
```

Here are ten reproducible draws.

```
set.seed(3)
rbinom(10, size = 2, prob = .7)

## [1] 2 1 2 2 1 1 2 2 1 1
```

Now generate 100,000 (i.e., `1e5`) reproducible dummy observations.

```
# how many would you like?
n_draws <- 1e5

set.seed(3)
d <- tibble(draws = rbinom(n_draws, size = 2, prob = .7))

d %>%
  group_by(draws) %>%
  count() %>%
  mutate(proportion = n / nrow(d))
```

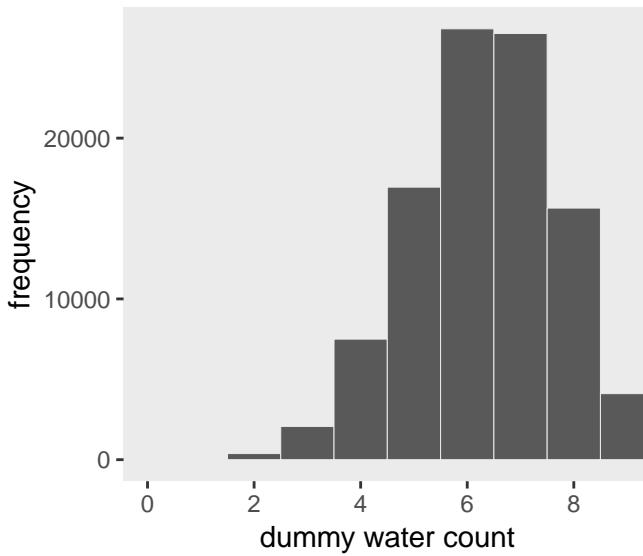
```
## # A tibble: 3 x 3
## # Groups:   draws [3]
##   draws     n proportion
##   <int> <int>      <dbl>
## 1 0     9000      0.09
## 2 1    42051      0.421
## 3 2    48949      0.489
```

As McElreath mused in the text, those simulated `proportion` values are very close to the analytically calculated values in our `density` column a few code blocks up.

Here's the simulation updated so $n = 9$, which we plot in our version of Figure 3.5.

```
set.seed(3)
d <- tibble(draws = rbinom(n_draws, size = 9, prob = .7))

# the histogram
d %>%
  ggplot(aes(x = draws)) +
  geom_histogram(binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("dummy water count",
                     breaks = seq(from = 0, to = 9, by = 2)) +
  ylab("frequency") +
  coord_cartesian(xlim = 0:9) +
  theme(panel.grid = element_blank())
```



McElreath suggested we play around with different values of `size` and `prob`. With the next block of code, we'll simulate nine conditions.

```
n_draws <- 1e5

simulate_binom <- function(n, probability){
  set.seed(3)
  rbinom(n_draws, size = n, prob = probability)
}

d <-
  tibble(n = c(3, 6, 9)) %>%
  expand(n, probability = c(.3, .6, .9)) %>%
  mutate(draws      = map2(n, probability, simulate_binom)) %>%
  ungroup() %>%
  mutate(n          = str_c("n = ", n),
         probability = str_c("p = ", probability)) %>%
  unnest()

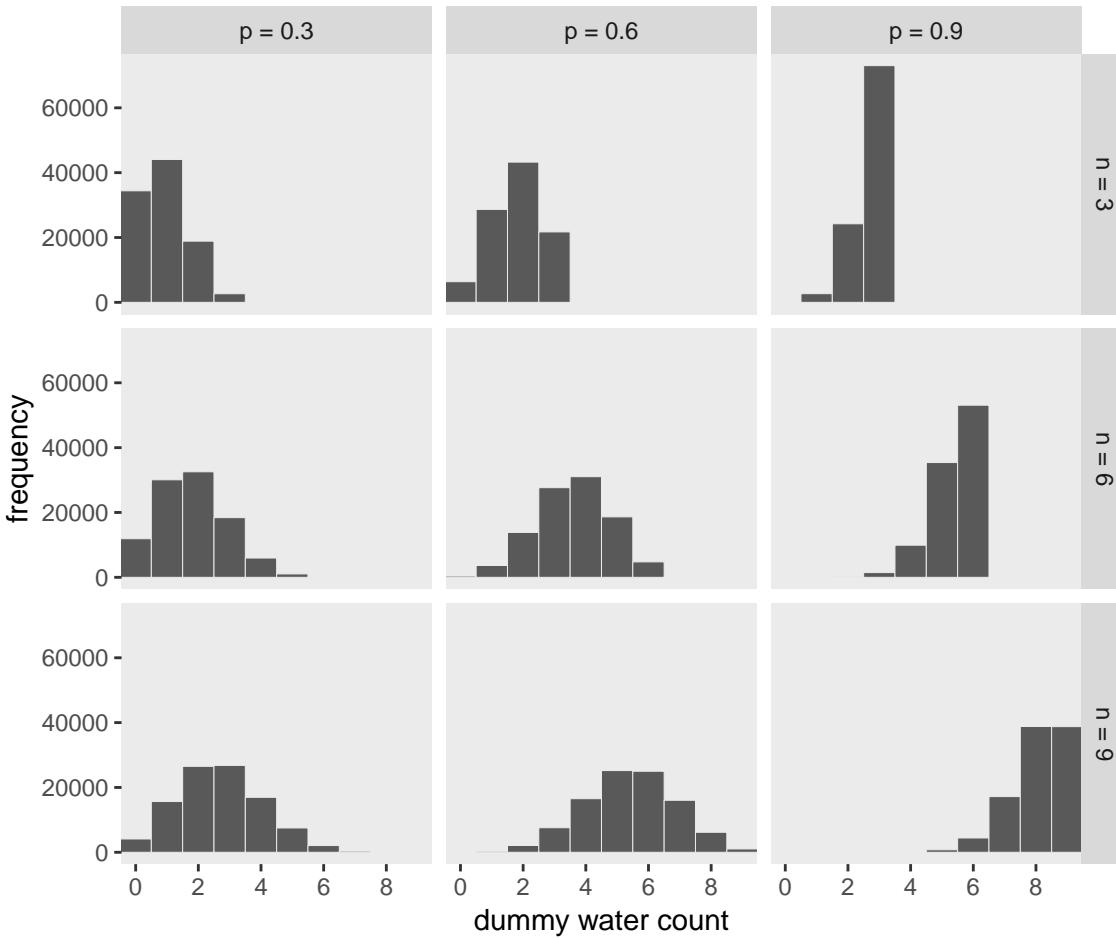
head(d)

## # A tibble: 6 x 3
##   n     probability draws
##   <chr> <chr>       <int>
## 1 n = 3 p = 0.3        0
## 2 n = 3 p = 0.3        2
## 3 n = 3 p = 0.3        1
## 4 n = 3 p = 0.3        0
## 5 n = 3 p = 0.3        1
## 6 n = 3 p = 0.3        1
```

The results look as follows:

```
d %>%
  ggplot(aes(x = draws)) +
  geom_histogram(binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("dummy water count",
                     breaks = seq(from = 0, to = 9, by = 2)) +
  ylab("frequency") +
  coord_cartesian(xlim = 0:9) +
```

```
theme(panel.grid = element_blank()) +
facet_grid(n ~ probability)
```



3.3.2 Model checking.

If you're new to applied statistics, you might be surprised how often mistakes arise.

3.3.2.1 Did the software work?

Let this haunt your dreams: "There is no way to really be sure that software works correctly" (p. 64).

If you'd like to dive deeper into these dark waters, check out one my favorite talks from StanCon 2018, [Esther Williams in the Harold Holt Memorial Swimming Pool](#), by the ineffable [Dan Simpson](#). If Simpson doesn't end up drowning you, see Gabry and Simpson's talk at the Royal Statistical Society 2018, [Visualization in Bayesian workflow](#), a follow-up blog [Maybe it's time to let the old ways die; or We broke R-hat so now we have to fix it](#), and that blog's associated pre-print by Vehtari, Gelman, Simpson, Carpenter, and Bürkner [Rank-normalization, folding, and localization: An improved R-hat for assessing convergence of MCMC](#).

3.3.2.2 Is the model adequate?

The implied predictions of the model are uncertain in two ways, and it's important to be aware of both.

First, there is observation uncertainty. For any unique value of the parameter p , there is a unique implied pattern of observations that the model expects. These patterns of observations are the same gardens of forking data that you explored in the previous chapter. These patterns are also what you sampled in the previous section. There is uncertainty in the predicted observations, because even if you know p with certainty, you won't know the next globe toss with certainty (unless $p = 0$ or $p = 1$).

Second, there is uncertainty about p . The posterior distribution over p embodies this uncertainty. And since there is uncertainty about p , there is uncertainty about everything that depends upon p . The uncertainty in p will interact with the sampling variation, when we try to assess what the model tells us about outcomes.

We'd like to *propagate* the parameter uncertainty—carry it forward—as we evaluate the implied predictions. All that is required is averaging over the posterior density for p , while computing the predictions. For each possible value of the parameter p , there is an implied distribution of outcomes. So if you were to compute the sampling distribution of outcomes at each value of p , then you could average all of these prediction distributions together, using the posterior probabilities of each value of p , to get a posterior predictive distribution. (p. 56, *emphasis* in the original)

All this is depicted in Figure 3.6. To get ready to make our version, let's first refresh our original grid approximation d .

```
# how many grid points would you like?
n <- 1001
n_success <- 6
n_trials <- 9

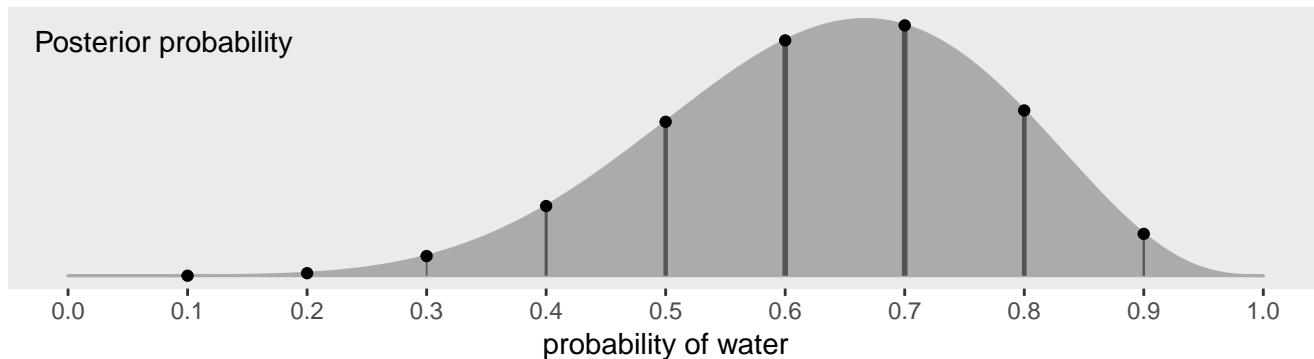
(
  d <-
    tibble(p_grid      = seq(from = 0, to = 1, length.out = n),
           # note we're still using a flat uniform prior
           prior       = 1) %>%
    mutate(likelihood = dbinom(n_success, size = n_trials, prob = p_grid)) %>%
    mutate(posterior  = (likelihood * prior) / sum(likelihood * prior))
)

## # A tibble: 1,001 x 4
##   p_grid prior likelihood posterior
##   <dbl> <dbl>     <dbl>     <dbl>
## 1 0        1     0.        0.
## 2 0.001    1     8.37e-17 8.37e-19
## 3 0.002    1     5.34e-15 5.34e-17
## 4 0.003    1     6.07e-14 6.07e-16
## 5 0.004    1     3.40e-13 3.40e-15
## 6 0.005    1     1.29e-12 1.29e-14
## 7 0.006    1     3.85e-12 3.85e-14
## 8 0.007    1     9.68e-12 9.68e-14
## 9 0.008    1     2.15e-11 2.15e-13
## 10 0.009   1     4.34e-11 4.34e-13
## # ... with 991 more rows
```

We can make our version of the top of Figure 3.6 with a little tricky filtering.

```
d %>%
  ggplot(aes(x = p_grid)) +
  geom_ribbon(aes(ymin = 0, ymax = posterior),
              color = "grey67", fill = "grey67") +
  geom_segment(data = . %>%
                  filter(p_grid %in% c(seq(from = .1, to = .9, by = .1), 3 / 10)),
              aes(xend = p_grid,
                  y = 0, yend = posterior, size = posterior),
              color = "grey33", show.legend = F) +
  geom_point(data = . %>%
                  filter(p_grid %in% c(seq(from = .1, to = .9, by = .1), 3 / 10)),
              aes(y = posterior)) +
  annotate(geom = "text",
          x = .08, y = .0025,
          label = "Posterior probability") +
  scale_size_continuous(range = c(0, 1)) +
```

```
scale_x_continuous("probability of water", breaks = c(0:10) / 10) +
scale_y_continuous(NULL, breaks = NULL) +
theme(panel.grid = element_blank())
```



Note how we weighted the widths of the vertical lines by the posterior density.

We'll need to do a bit of wrangling before we're ready to make the plot in the middle panel of Figure 3.6.

```
n_draws <- 1e5

simulate_binom <- function(probability){
  set.seed(3)
  rbinom(n_draws, size = 9, prob = probability)
}

d_small <-
  tibble(probability = seq(from = .1, to = .9, by = .1)) %>%
  mutate(draws      = purrr::map(probability, simulate_binom)) %>%
  unnest(draws) %>%
  mutate(label       = str_c("p = ", probability))

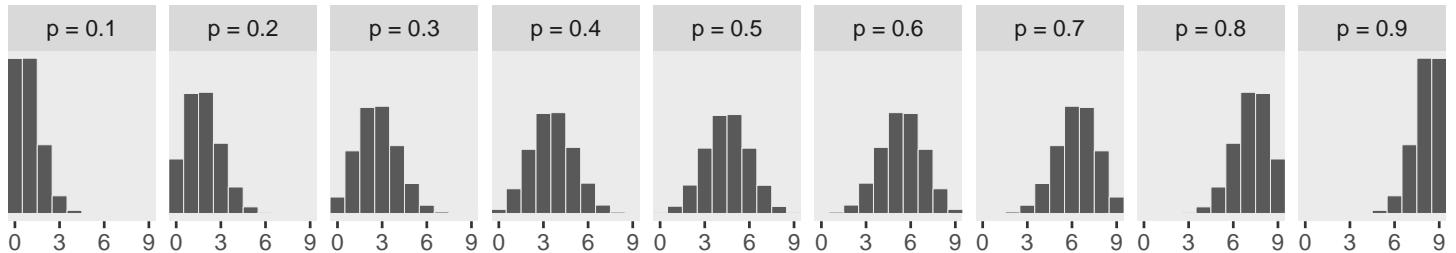
head(d_small)

## # A tibble: 6 x 3
##   probability draws label
##       <dbl>    <int> <chr>
## 1         0.1      0 p = 0.1
## 2         0.1      2 p = 0.1
## 3         0.1      0 p = 0.1
## 4         0.1      0 p = 0.1
## 5         0.1      1 p = 0.1
## 6         0.1      1 p = 0.1
```

Now we're ready to plot.

```
d_small %>%
  ggplot(aes(x = draws)) +
  geom_histogram(binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous(NULL, breaks = seq(from = 0, to = 9, by = 3)) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(subtitle = "Sampling distributions") +
  coord_cartesian(xlim = 0:9) +
  theme(panel.grid = element_blank()) +
  facet_wrap(~ label, ncol = 9)
```

Sampling distributions



To make the plot at the bottom of Figure 3.6, we'll redefine our `samples`, this time including the `w` variable (see the R code 3.26 block in the text).

```
# how many samples would you like?
n_samples <- 1e4

# make it reproducible
set.seed(3)

samples <-
  d %>%
  sample_n(size = n_samples, weight = posterior, replace = T) %>%
  mutate(w = purrr::map_dbl(p_grid, rbinom, n = 1, size = 9))

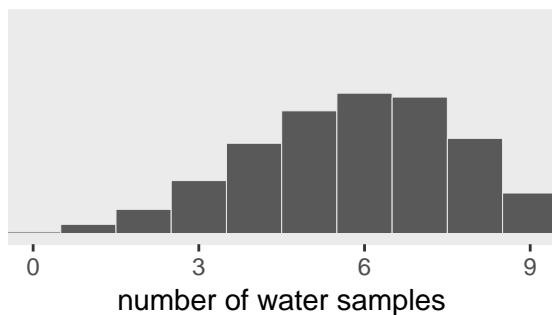
glimpse(samples)

## Observations: 10,000
## Variables: 5
## $ p_grid      <dbl> 0.734, 0.808, 0.385, 0.328, 0.510, 0.511, 0.756, 0.674, 0.578, 0.520, 0.512, ...
## $ prior       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ likelihood  <dbl> 0.24722965, 0.16544404, 0.06363104, 0.03174170, 0.17389560, 0.17487642, 0.22...
## $ posterior   <dbl> 0.0024722965, 0.0016544404, 0.0006363104, 0.0003174170, 0.0017389560, 0.0017...
## $ w           <dbl> 6, 7, 7, 2, 8, 3, 8, 6, 7, 6, 3, 5, 5, 5, 7, 7, 9, 6, 7, 2, 7, 9, 6, 6, 8, 4...
```

Here's our histogram.

```
samples %>%
  ggplot(aes(x = w)) +
  geom_histogram(binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("number of water samples",
                     breaks = seq(from = 0, to = 9, by = 3)) +
  scale_y_continuous(NULL, breaks = NULL) +
  ggtitle("Posterior predictive distribution") +
  coord_cartesian(xlim = 0:9,
                  ylim = 0:3000) +
  theme(panel.grid = element_blank())
```

Posterior predictive distribution



In Figure 3.7, McElreath considered the longest sequence of the sample values. We've been using `rbinom()` with the size parameter set to 9 for our simulations. E.g.,

```
rbinom(10, size = 9, prob = .6)
```

```
## [1] 6 8 6 5 6 7 6 5 3 6
```

Notice this collapses (i.e., aggregated) over the sequences within the individual sets of 9. What we need is to simulate nine individual trials many times over. For example, this

```
rbinom(9, size = 1, prob = .6)
```

```
## [1] 1 1 1 1 1 1 1 1 0 0
```

would be the disaggregated version of just one of the numerals returned by `rbinom()` when `size = 9`. So let's try simulating again with un-aggregated samples. We'll keep adding to our `samples` tibble. In addition to the disaggregated `draws` based on the p values listed in `p_grid`, we'll also want to add a row index for each of those `p_grid` values—it'll come in handy when we plot.

```
# make it reproducible
set.seed(3)

samples <-
  samples %>%
  mutate(iter = 1:n(),
        draws = purrr::map(p_grid, rbinom, n = 9, size = 1)) %>%
  unnest(draws)

glimpse(samples)

## Observations: 90,000
## Variables: 7
## $ p_grid      <dbl> 0.734, 0.734, 0.734, 0.734, 0.734, 0.734, 0.734, 0.734, 0.808, 0.808, ...
## $ prior       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
## $ likelihood  <dbl> 0.24722965, 0.24722965, 0.24722965, 0.24722965, 0.24722965, 0.24722965, 0.24...
## $ posterior   <dbl> 0.0024722965, 0.0024722965, 0.0024722965, 0.0024722965, 0.0024722965, 0.0024...
## $ w           <dbl> 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7...
## $ iter        <int> 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3...
## $ draws       <int> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1...
```

The main action is in the `draws` column.

Now we have to count the longest sequences. The base R `rle()` function will help with that. Consider McElreath's sequence of tosses.

```
tosses <- c("w", "l", "w", "w", "w", "l", "w", "l", "w")
```

You can plug that into `rle()`.

```
rle(tosses)
```

```
## Run Length Encoding
##  lengths: int [1:7] 1 1 3 1 1 1 1
##  values : chr [1:7] "w" "l" "w" "l" "w" "l" "w"
```

For our purposes, we're interested in `lengths`. That tells us the length of each sequences of the same value. The 3 corresponds to our run of three ws. The `max()` function will help us confirm it's the largest value.

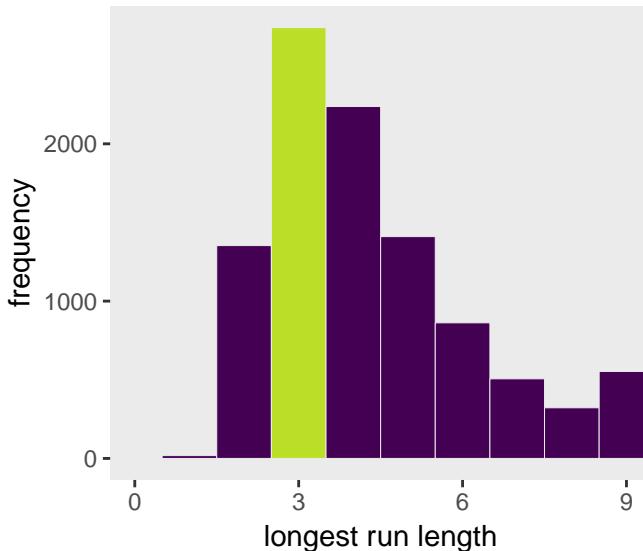
```
rle(tosses)$lengths %>% max()
```

```
## [1] 3
```

Now let's apply our method to the data and plot.

```
samples %>%
  group_by(iter) %>%
  summarise(longest_run_length = rle(draws)$lengths %>% max()) %>%

  ggplot(aes(x = longest_run_length)) +
  geom_histogram(aes(fill = longest_run_length == 3),
                 binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("longest run length",
                     breaks = seq(from = 0, to = 9, by = 3)) +
  scale_fill_viridis_d(option = "D", end = .9) +
  ylab("frequency") +
  coord_cartesian(xlim = 0:9) +
  theme(panel.grid = element_blank(),
        legend.position = "none")
```



Let's look at `rle()` again.

```
rle(tosses)
```

```
## Run Length Encoding
##   lengths: int [1:7] 1 1 3 1 1 1 1
##   values : chr [1:7] "w" "l" "w" "l" "w" "l" "w"
```

We can use the length of the output (i.e., 7 in this example) as the numbers of switches from, in this case, “w” and “l”.

```
rle(tosses)$lengths %>% length()
```

```
## [1] 7
```

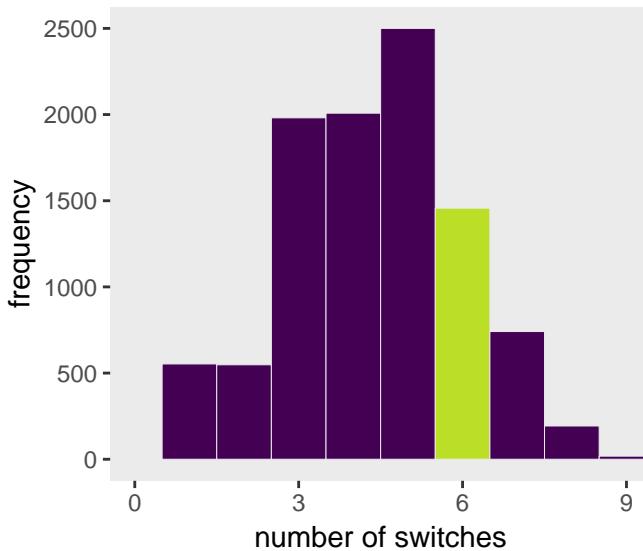
With that new trick, we're ready to make the right panel of Figure 3.7.

```

samples %>%
  group_by(iter) %>%
  summarise(longest_run_length = rle(draws)$lengths %>% length()) %>%

  ggplot(aes(x = longest_run_length)) +
  geom_histogram(aes(fill = longest_run_length == 6),
                 binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("number of switches",
                     breaks = seq(from = 0, to = 9, by = 3)) +
  scale_fill_viridis_d(option = "D", end = .9) +
  ylab("frequency") +
  coord_cartesian(xlim = 0:9) +
  theme(panel.grid = element_blank(),
        legend.position = "none")

```



3.4 Summary Let's practice in brms

Open brms.

```
library(brms)
```

In brms, we'll fit the primary model of $w = 6$ and $n = 9$ much like we did at the end of the project for Chapter 2.

```

b3.1 <-
  brm(data = list(w = 6),
       family = binomial(link = "identity"),
       w | trials(9) ~ 1,
       # this is a flat prior
       prior(beta(1, 1), class = Intercept),
       seed = 3,
       control = list(adapt_delta = .999))

```

We'll learn more about the beta distribution in Chapter 11. But for now, here's the posterior summary for `b_Intercept`, the probability of a "w".

```

posterior_summary(b3.1)[["b_Intercept", ] %>%
  round(digits = 2)

```

```
## #> #> #> #>
```

Estimate	Est.Error	Q2.5	Q97.5
0.64	0.14	0.36	0.88

As we'll fully cover in the next chapter, `Estimate` is the posterior mean, the two `Q` columns are the quantile-based 95% intervals, and `Est.Error` is the posterior standard deviation.

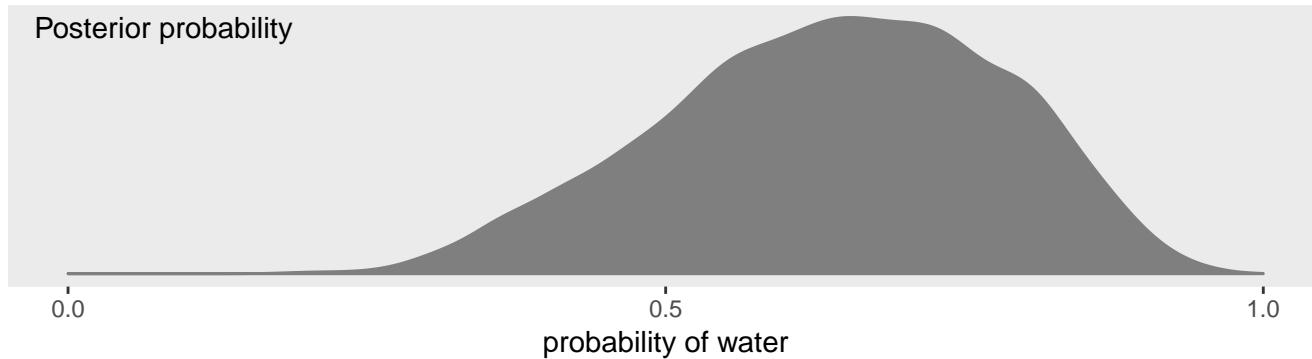
Much like the way we used the `samples()` function to simulate probability values, above, we can do so with `fitted()` within the brms framework. But we will have to specify `scale = "linear"` in order to return results in the probability metric. By default, `brms::fitted()` will return summary information. Since we want actual simulation draws, we'll specify `summary = F`.

```
f <-  
  fitted(b3.1, summary = F,  
          scale = "linear") %>%  
  as_tibble() %>%  
  set_names("p")  
  
glimpse(f)  
  
## #> #> #> #>  
## #> #> #> #>  
## #> #> #> #>
```

Observations: 4,000
Variables: 1
\$ p <dbl> 0.6920484, 0.5559454, 0.6096088, 0.5305334, 0.4819733, 0.6724561, 0.6402367, 0.835656...

By default, we have a generically-named vector `V1` of 4000 samples. We'll explain the defaults in later chapters. For now, notice we can view these in a density.

```
f %>%  
  ggplot(aes(x = p)) +  
  geom_density(fill = "grey50", color = "grey50") +  
  annotate(geom = "text",  
          x = .08, y = 2.5,  
          label = "Posterior probability") +  
  scale_x_continuous("probability of water",  
                     breaks = c(0, .5, 1),  
                     limits = 0:1) +  
  scale_y_continuous(NULL, breaks = NULL) +  
  theme(panel.grid = element_blank())
```



Looks a lot like the posterior probability density at the top of Figure 3.6, doesn't it? Much like we did with `samples`, we can use this distribution of probabilities to predict histograms of `w` counts. With those in hand, we can make an analogue to the histogram in the bottom panel of Figure 3.6.

```
# the simulation  
set.seed(3)  
  
f <-  
  f %>%
```

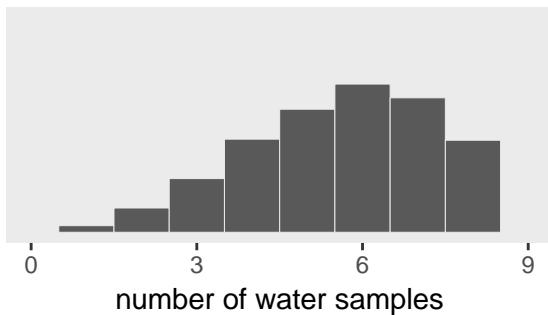
```

mutate(w = rbinom(n(), size = n_trials, prob = p))

# the plot
f %>%
  ggplot(aes(x = w)) +
  geom_histogram(binwidth = 1, center = 0,
                 color = "grey92", size = 1/10) +
  scale_x_continuous("number of water samples",
                     breaks = seq(from = 0, to = 9, by = 3), limits = c(0, 9)) +
  scale_y_continuous(NULL, breaks = NULL, limits = c(0, 1200)) +
  ggtitle("Posterior predictive distribution") +
  theme(panel.grid = element_blank())

```

Posterior predictive distribution



As you might imagine, we can use the output from `fitted()` to return disaggregated batches of 0s and 1s, too. And we could even use those disaggregated 0s and 1s to examine longest run lengths and numbers of switches as in the analyses for Figure 3.7. I'll leave those as exercises for the interested reader.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```

sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats      graphics   grDevices   utils      datasets   methods    base
##
## other attached packages:
## [1] brms_2.8.8      Rcpp_1.0.1       tidybayes_1.0.4 forcats_0.3.0  stringr_1.4.0   dplyr_0.8.0.1
## [7] purrr_0.2.5     readr_1.1.1     tidyverse_1.2.1
## [9]

```

```
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2          ggridges_0.5.0           rsconnect_0.8.8
## [4] rprojroot_1.3-2          ggstance_0.3             markdown_0.8
## [7] base64enc_0.1-3          rethinking_1.80          rstudioapi_0.7
## [10] rstan_2.18.2            svUnit_0.7-12           DT_0.4
## [13] fansi_0.4.0              mvtnorm_1.0-10          lubridate_1.7.4
## [16] xml2_1.2.0               bridgesampling_0.6-0    knitr_1.20
## [19] shinythemes_1.1.1         bayesplot_1.6.0          jsonlite_1.5
## [22] LaplacesDemon_16.1.1     broom_0.5.1              shiny_1.1.0
## [25] compiler_3.5.1           httr_1.3.1              backports_1.1.4
## [28] assertthat_0.2.0         Matrix_1.2-14           lazyeval_0.2.2
## [31] cli_1.0.1                later_0.7.3              htmltools_0.3.6
## [34] prettyunits_1.0.2        tools_3.5.1              igraph_1.2.1
## [37] coda_0.19-2              gtable_0.3.0             glue_1.3.1.9000
## [40] reshape2_1.4.3            cellranger_1.1.0         nlme_3.1-137
## [43] crosstalk_1.0.0          xfun_0.3                 ps_1.2.1
## [46] rvest_0.3.2              mime_0.5                miniUI_0.1.1.1
## [49] gtools_3.8.1              MASS_7.3-50              zoo_1.8-2
## [52] scales_1.0.0              colourpicker_1.0         hms_0.4.2
## [55] promises_1.0.1           Brobdingnag_1.2-6        parallel_3.5.1
## [58] inline_0.3.15             shinystan_2.5.0          yaml_2.1.19
## [61] gridExtra_2.3              loo_2.1.0                StanHeaders_2.18.0-1
## [64] stringi_1.4.3             dygraphs_1.1.1.5         pkgbuild_1.0.2
## [67] rlang_0.3.4               pkgconfig_2.0.2          matrixStats_0.54.0
## [70] HDInterval_0.2.0          evaluate_0.10.1         lattice_0.20-35
## [73] rstantools_1.5.1          htmlwidgets_1.2          labeling_0.3
## [76] processx_3.2.1            tidyselect_0.2.5         plyr_1.8.4
## [79] magrittr_1.5               bookdown_0.9             R6_2.3.0
## [82] generics_0.0.2             pillar_1.3.1             haven_1.1.2
## [85] withr_2.1.2               xts_0.10-2              abind_1.4-5
## [88] modelr_0.1.2              crayon_1.3.4             arrayhelpers_1.0-20160527
## [91] utf8_1.1.4                rmarkdown_1.10            grid_3.5.1
## [94] readxl_1.1.0              callr_3.1.0              threejs_0.3.1
## [97] digest_0.6.18             xtable_1.8-2             httpuv_1.4.4.2
## [100] stats4_3.5.1             munsell_0.5.0            viridisLite_0.3.0
## [103] shinyjs_1.0
```


Chapter 4

Linear Models

Linear regression is the geocentric model of applied statistics. By “linear regression”, we will mean a family of simple statistical golems that attempt to learn about the mean and variance of some measurement, using an additive combination of other measurements. Like geocentrism, linear regression can usefully describe a very large variety of natural phenomena. Like geocentrism, linear is a descriptive model that corresponds to many different process models. If we read its structure too literally, we’re likely to make mistakes. But used wisely, these little linear golems continue to be useful. (p. 71)

4.1 Why normal distributions are normal

After laying out his soccer field coin toss shuffle premise, McElreath wrote:

It’s hard to say where any individual person will end up, but you can say with great confidence what the collection of positions will be. The distances will be distributed in approximately normal, or Gaussian, fashion. This is true even though the underlying distribution is binomial. It does this because there are so many more possible ways to realize a sequence of left-right steps that sums to zero. There are slightly fewer ways to realize a sequence that ends up one step left or right of zero, and so on, with the number of possible sequences declining in the characteristic bell curve of the normal distribution. (p. 72)

4.1.1 Normal by addition.

Here’s a way to do the simulation necessary for the plot in the top panel of Figure 4.2.

```
library(tidyverse)

# we set the seed to make the results of `runif()` reproducible.
set.seed(4)
pos <-
  replicate(100, runif(16, -1, 1)) %>%
    as_tibble() %>%
    rbind(0, .) %>%
    mutate(step = 0:16) %>%
    gather(key, value, -step) %>%
    mutate(person = rep(1:100, each = 17)) %>%
    # the next two lines allow us to make cumulative sums within each person
    group_by(person) %>%
    mutate(position = cumsum(value)) %>%
    ungroup() # ungrouping allows for further data manipulation
```

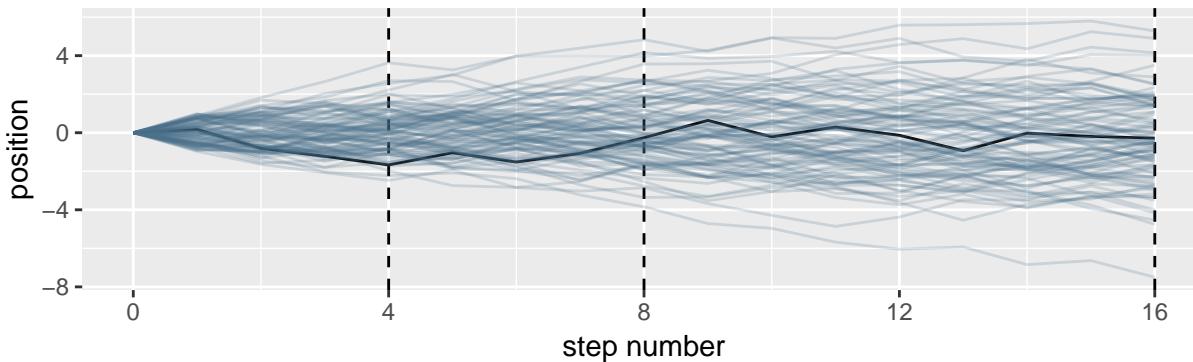
We might `glimpse()` at the data.

```
glimpse(pos)

## Observations: 1,700
## Variables: 5
## $ step      <int> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 0, 1, 2, 3, 4, 5, 6, ...
## $ key       <chr> "V1", ...
## $ value     <dbl> 0.0000000, 0.17160061, -0.98210841, -0.41252078, -0.44525008, 0.62714843, -0. ....
## $ person    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ position   <dbl> 0.0000000, 0.17160061, -0.81050780, -1.22302857, -1.66827866, -1.04113023, -1. ....
```

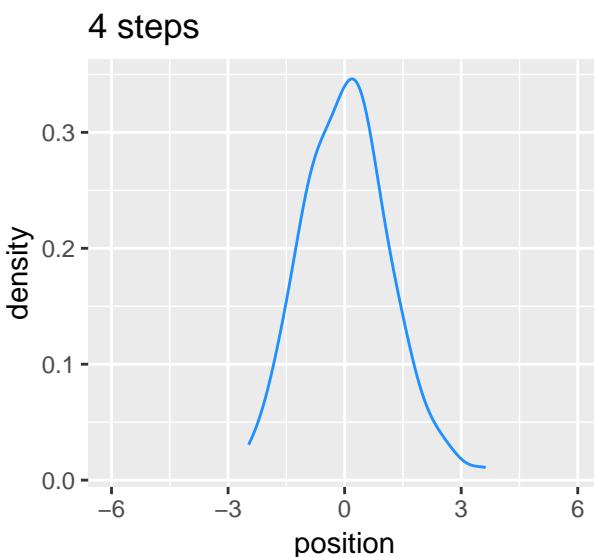
And here's the actual plot code.

```
ggplot(data = pos,
       aes(x = step, y = position, group = person)) +
  geom_vline(xintercept = c(4, 8, 16), linetype = 2) +
  geom_line(aes(color = person < 2, alpha = person < 2)) +
  scale_color_manual(values = c("skyblue4", "black")) +
  scale_alpha_manual(values = c(1/5, 1)) +
  scale_x_continuous("step number", breaks = c(0, 4, 8, 12, 16)) +
  theme(legend.position = "none")
```



Here's the code for the bottom three plots of Figure 4.2.

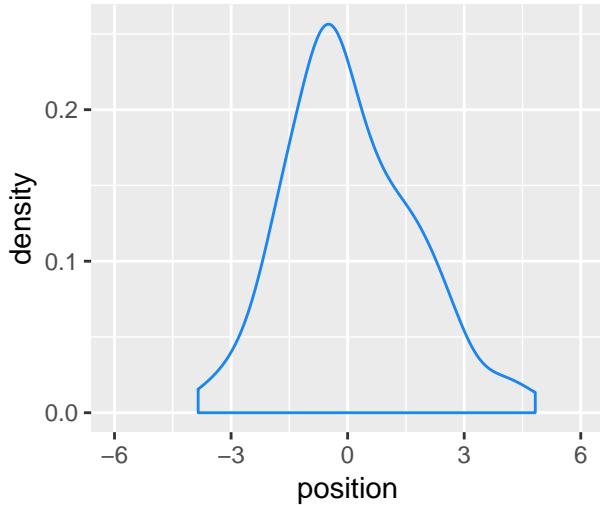
```
# Figure 4.2.a.
pos %>%
  filter(step == 4) %>%
  ggplot(aes(x = position)) +
  geom_line(stat = "density", color = "dodgerblue1") +
  coord_cartesian(xlim = -6:6) +
  labs(title = "4 steps")
```



```
# Figure 4.2.b.
```

```
pos %>%
  filter(step == 8) %>%
  ggplot(aes(x = position)) +
  geom_density(color = "dodgerblue2") +
  coord_cartesian(xlim = -6:6) +
  labs(title = "8 steps")
```

8 steps



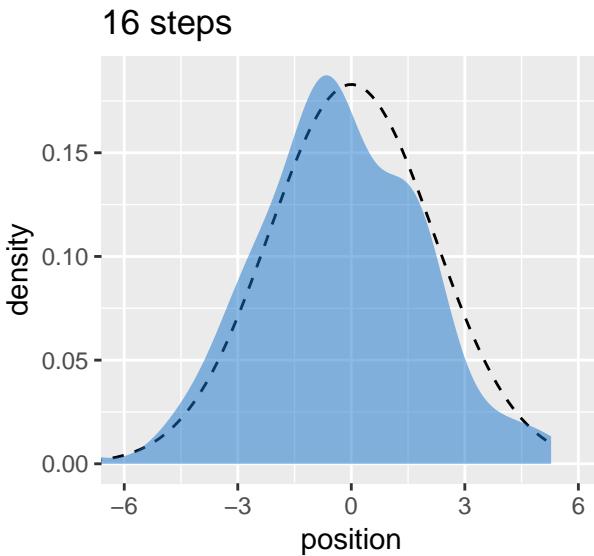
```
# this is an intermediary step to get an SD value
```

```
pos %>%
  filter(step == 16) %>%
  summarise(sd = sd(position))
```

```
## # A tibble: 1 x 1
##       sd
##   <dbl>
## 1  2.18
```

```
# Figure 4.2.c.
```

```
pos %>%
  filter(step == 16) %>%
  ggplot(aes(x = position)) +
  stat_function(fun = dnorm,
                args = list(mean = 0, sd = 2.180408),
                linetype = 2) + # 2.180408 came from the previous code block
  geom_density(color = "transparent", fill = "dodgerblue3", alpha = 1/2) +
  coord_cartesian(xlim = -6:6) +
  labs(title = "16 steps",
       y      = "density")
```



While we were at it, we explored a few ways to express densities. The main action was with the `geom_line()`, `geom_density()`, and `stat_function()` functions.

4.1.2 Normal by multiplication.

Here's McElreath's simple random growth rate.

```
set.seed(4)
prod(1 + runif(12, 0, 0.1))
```

```
## [1] 1.774719
```

In the `runif()` part of that code, we generated 12 random draws from the uniform distribution with bounds $[0, 0.1]$. Within the `prod()` function, we first added 1 to each of those values and then computed their product. Consider a more explicit variant of the code.

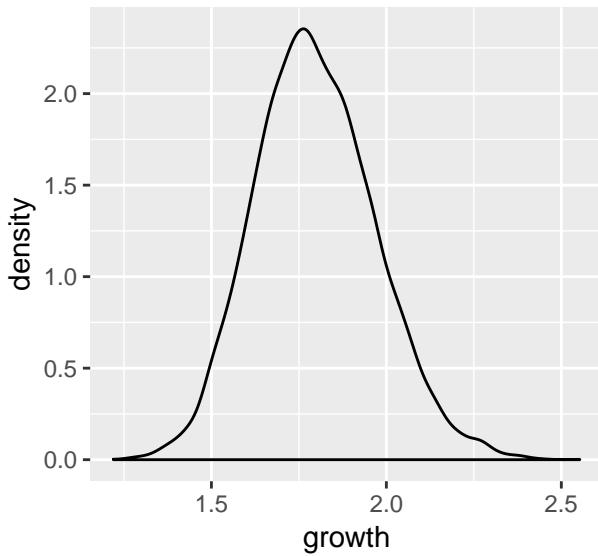
```
set.seed(4)
tibble(a = 1,
       b = runif(12, 0, 0.1)) %>%
  mutate(c = a + b) %>%
  summarise(p = prod(c))
```

```
## # A tibble: 1 x 1
##       p
##   <dbl>
## 1 1.77
```

Same result. Rather than using base R `replicate()` to do this many times, let's practice with `purrr::map_dbl()` instead (see [here](#) for details).

```
set.seed(4)
growth <-
  tibble(growth = map_dbl(1:10000, ~ prod(1 + runif(12, 0, 0.1))))
```

```
ggplot(data = growth, aes(x = growth)) +
  geom_density()
```

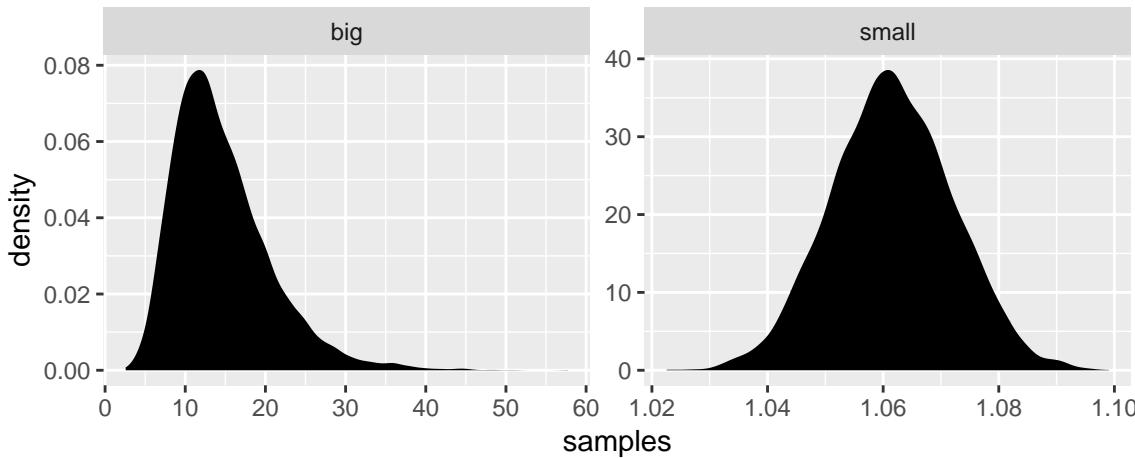


"The smaller the effect of each locus, the better this additive approximation will be" (p. 74). Let's compare big and small.

```
# simulate
set.seed(4)

samples <-
  tibble(big    = map_dbl(1:10000, ~ prod(1 + runif(12, 0, 0.5))),
         small = map_dbl(1:10000, ~ prod(1 + runif(12, 0, 0.01)))) %>%
# wrangle
gather(distribution, samples)

# plot
samples %>%
  ggplot(aes(x = samples)) +
  geom_density(fill = "black", color = "transparent") +
  facet_wrap(~distribution, scales = "free")
```



Yep, the `small` samples were more Gaussian.

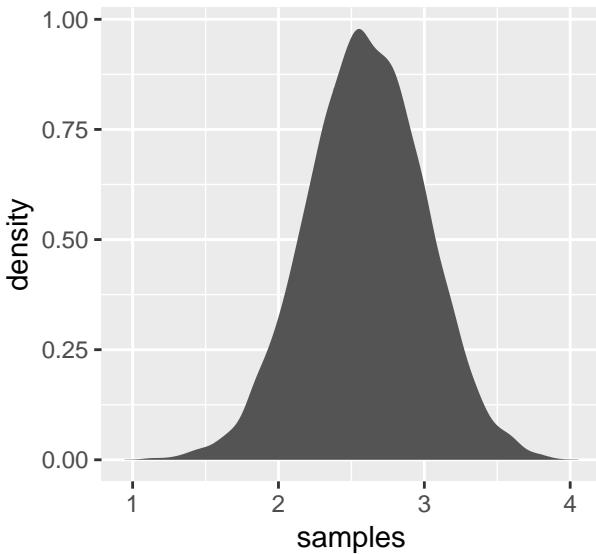
4.1.3 Normal by log-multiplication.

Instead of saving our tibble, we'll just feed it directly into our plot.

```
set.seed(4)

tibble(samples = map_dbl(1:1e4, ~ log(prod(1 + runif(12, 0, 0.5))))) %>%
```

```
ggplot(aes(x = samples)) +
  geom_density(color = "transparent",
              fill = "gray33")
```



What we did was really compact. Walking it out a bit, here's what we all did within the second argument within `map_dbl()` (i.e., everything within `log()`).

```
tibble(a = runif(12, 0, 0.5),
      b = 1) %>%
  mutate(c = a + b) %>%
  summarise(p = prod(c) %>% log())

## # A tibble: 1 x 1
##       p
##   <dbl>
## 1  2.82
```

And based on the first argument within `map_dbl()`, we did that 10,000 times, after which we converted the results to a tibble and then fed those data into `ggplot2`.

4.1.4 Using Gaussian distributions.

I really like the justifications in the following subsections.

4.1.4.1 Ontological justification.

The Gaussian is

a widespread pattern, appearing again and again at different scales and in different domains. Measurement errors, variations in growth, and the velocities of molecules all tend towards Gaussian distributions. These processes do this because at their heart, these processes add together fluctuations. And repeatedly adding finite fluctuations results in a distribution of sums that have shed all information about the underlying process, aside from mean and spread.

One consequence of this is that statistical models based on Gaussian distributions cannot reliably identify micro-process... (p. 75)

But they can still be useful.

4.1.4.2 Epistemological justification.

Another route to justifying the Gaussian as our choice of skeleton, and a route that will help us appreciate later why it is often a poor choice, is that it represents a particular state of ignorance. When all we know or are willing to say about a distribution of measures (measures are continuous values on the real number line) is their mean and variance, then the Gaussian distribution arises as the most consistent with our assumptions.

That is to say that the Gaussian distribution is the most natural expression of our state of ignorance, because if all we are willing to assume is that a measure has finite variance, the Gaussian distribution is the shape that can be realized in the largest number of ways and does not introduce any new assumptions. It is the least surprising and least informative assumption to make. In this way, the Gaussian is the distribution most consistent with our assumptions... If you don't think the distribution should be Gaussian, then that implies that you know something else that you should tell your golem about, something that would improve inference. (pp. 75–76)

In the **Overthinking: Gaussian distribution** box that follows, McElreath gave the formula. Let y be the criterion, μ be the mean, and σ be the standard deviation. Then the probability density of some Gaussian value y is

$$p(y|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$$

4.2 A language for describing models

Our mathy ways of summarizing models will be something like

$$\begin{aligned} \text{criterion}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \beta \times \text{predictor}_i \\ \beta &\sim \text{Normal}(0, 10) \\ \sigma &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

And as McElreath then followed up with, “If that doesn’t make much sense, good. That indicates that you are holding the right textbook” (p. 77). Welcome applied statistics!

4.2.1 Re-describing the globe tossing model.

For the globe tossing model, the probability p of a count of water w based on n trials was

$$\begin{aligned} w &\sim \text{Binomial}(n, p) \\ p &\sim \text{Uniform}(0, 1) \end{aligned}$$

We can break McElreath’s R code 4.6 down a little bit with a tibble like so.

```
# how many `p_grid` points would you like?
n_points <- 100

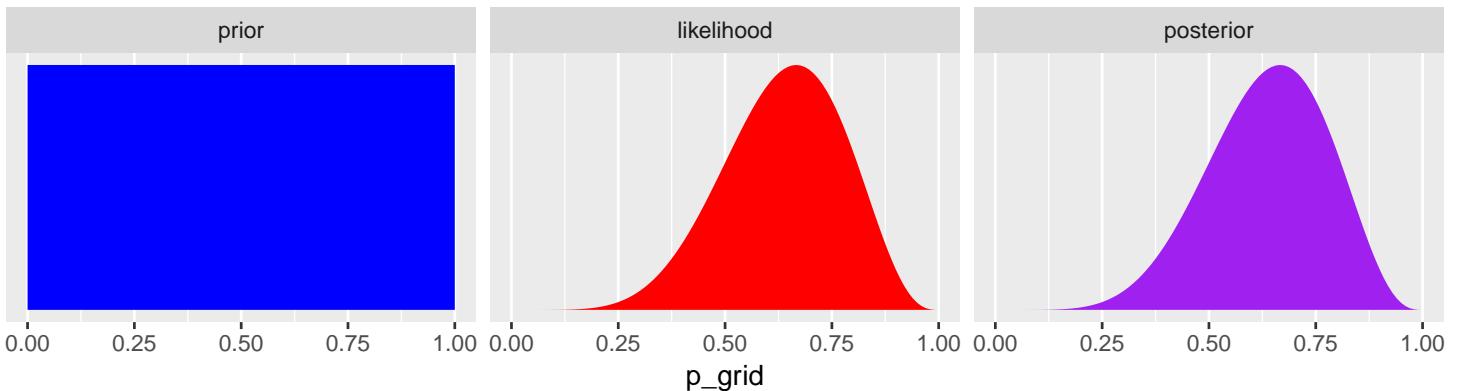
d <-
  tibble(w      = 6,
        n      = 9,
        p_grid = seq(from = 0, to = 1, length.out = n_points)) %>%
  mutate(prior   = dunif(p_grid, 0, 1),
        likelihood = dbinom(w, n, p_grid)) %>%
  mutate(posterior = likelihood * prior / sum(likelihood * prior))

head(d)
```

```
## # A tibble: 6 x 6
##       w     n p_grid prior likelihood posterior
##   <dbl> <dbl> <dbl>    <dbl>      <dbl>
## 1     6     9 0        1     0.        0.
## 2     6     9 0.0101   1     8.65e-11  8.74e-12
## 3     6     9 0.0202   1     5.37e- 9  5.43e-10
## 4     6     9 0.0303   1     5.93e- 8  5.99e- 9
## 5     6     9 0.0404   1     3.23e- 7  3.26e- 8
## 6     6     9 0.0505   1     1.19e- 6  1.21e- 7
```

In case you were curious, here's what they look like:

```
d %>%
  select(-w, -n) %>%
  gather(key, value, -p_grid) %>%
  # this line allows us to dictate the order the panels will appear in
  mutate(key = factor(key, levels = c("prior", "likelihood", "posterior"))) %>%
  ggplot(aes(x = p_grid, ymin = 0, ymax = value, fill = key)) +
  geom_ribbon() +
  scale_fill_manual(values = c("blue", "red", "purple")) +
  scale_y_continuous(NULL, breaks = NULL) +
  theme(legend.position = "none") +
  facet_wrap(~key, scales = "free")
```



The posterior is a combination of the prior and the likelihood. And when the prior is flat across the parameter space, the posterior is just the likelihood re-expressed as a probability. As we go along, you'll see that we almost never use flat priors.

4.3 A Gaussian model of height

There are an infinite number of possible Gaussian distributions. Some have small means. Others have large means. Some are wide, with a large σ . Others are narrow. We want our Bayesian machine to consider every possible distribution, each defined by a combination of μ and σ , and rank them by posterior plausibility. (p. 79)

4.3.1 The data.

Let's get the data from McElreath's [rethinking package](#).

```
library(rethinking)
data(Howell1)
d <- Howell1
```

Here we open our main statistical package, Bürkner's [brms](#). But before we do, we'll need to detach the rethinking package. R will not allow users to use a function from one package that shares the same name as a different function from another

package if both packages are open at the same time. The `rethinking` and `brms` packages are designed for similar purposes and, unsurprisingly, overlap in the names of their functions. To prevent problems, we will always make sure `rethinking` is detached before using `brms`. To learn more on the topic, see [this R-bloggers post](#).

```
rm(Howell1)
detach(package:rethinking, unload = T)
library(brms)

## Warning: package 'Rcpp' was built under R version 3.5.2
```

Go ahead and investigate the data with `str()`, the tidyverse analogue for which is `glimpse()`.

```
d %>%
  str()

## 'data.frame':    544 obs. of  4 variables:
## $ height: num  152 140 137 157 145 ...
## $ weight: num  47.8 36.5 31.9 53 41.3 ...
## $ age   : num  63 63 65 41 51 35 32 27 19 54 ...
## $ male  : int  1 0 0 1 0 1 0 1 0 1 ...
```

Here are the `height` values.

```
d %>%
  select(height) %>%
  head()
```

```
##    height
## 1 151.765
## 2 139.700
## 3 136.525
## 4 156.845
## 5 145.415
## 6 163.830
```

We can use `filter()` to make an adults-only data frame.

```
d2 <-
d %>%
  filter(age >= 18)
```

4.3.1.1 Overthinking: Data frames.

This probably reflects my training history, but the structure of a data frame seems natural and inherently appealing, to me. So I can't relate to the "annoying" comment. But if you're in the other camp, do check out either of these two data wrangling talks ([here](#) and [here](#)) by the ineffable [Jenny Bryan](#).

4.3.1.2 Overthinking: Index magic.

For more on indexing, check out [chapter 9](#) of Roger Peng's *R Programming for Data Science* or even the [Subsetting](#) subsection from *R4DS*.

4.3.2 The model.

The likelihood for our model is

$$h_i \sim \text{Normal}(\mu, \sigma)$$

Our μ prior will be

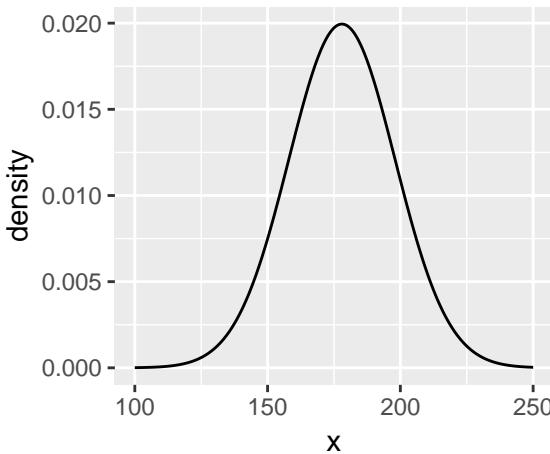
$$\mu \sim \text{Normal}(178, 20)$$

And our prior for σ will be

$$\sigma \sim \text{Uniform}(0, 50)$$

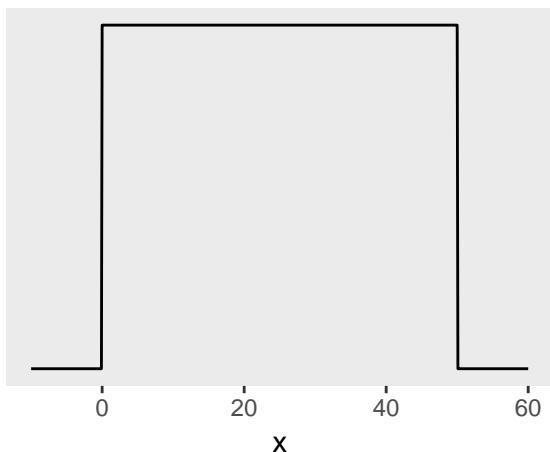
Here's the shape of the prior for μ in $N(178, 20)$.

```
ggplot(data = tibble(x = seq(from = 100, to = 250, by = .1)),
       aes(x = x, y = dnorm(x, mean = 178, sd = 20))) +
  geom_line() +
  ylab("density")
```



And here's the ggplot2 code for our prior for σ , a uniform distribution with a minimum value of 0 and a maximum value of 50. We don't really need the y axis when looking at the shapes of a density, so we'll just remove it with `scale_y_continuous()`.

```
tibble(x = seq(from = -10, to = 60, by = .1)) %>%
  ggplot(aes(x = x, y = dunif(x, min = 0, max = 50))) +
  geom_line() +
  scale_y_continuous(NULL, breaks = NULL) +
  theme(panel.grid = element_blank())
```



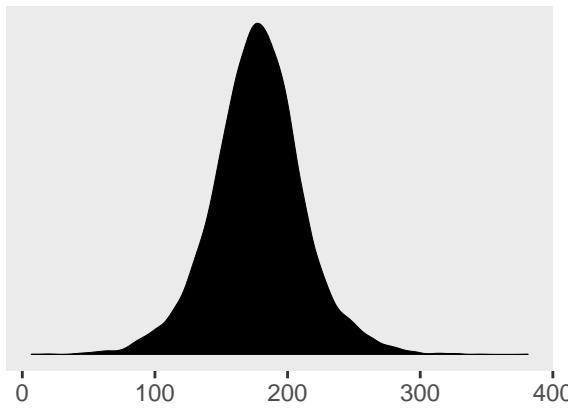
We can simulate from both priors at once to get a prior probability distribution of `heights`.

```
n <- 1e4

set.seed(4)
tibble(sample_mu      = rnorm(n, mean = 178,           sd = 20),
       sample_sigma = runif(n, min = 0,             max = 50)) %>%
  mutate(x          = rnorm(n, mean = sample_mu, sd = sample_sigma)) %>%

ggplot(aes(x = x)) +
  geom_density(fill = "black", size = 0) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(subtitle = expression(paste("Prior predictive distribution for ", italic(h[i]))),
       x       = NULL) +
  theme(panel.grid = element_blank())
```

Prior predictive distribution for h_i



As McElreath wrote, we've made a “vaguely bell-shaped density with thick tails. It is the expected distribution of heights, averaged over the prior” (p. 83).

4.3.3 Grid approximation of the posterior distribution.

As McElreath explained, you'll never use this for practical data analysis. But I found this helped me better understanding what exactly we're doing with Bayesian estimation. So let's play along.

```
n <- 200

d_grid <-
  tibble(mu     = seq(from = 140, to = 160, length.out = n),
        sigma = seq(from = 4,   to = 9,    length.out = n)) %>%
  # we'll accomplish with `tidy::expand()` what McElreath did with base R `expand.grid()`
  expand(mu, sigma)

head(d_grid)
```

```
## # A tibble: 6 x 2
##       mu   sigma
##   <dbl> <dbl>
## 1 140    4.00
## 2 140    4.03
## 3 140    4.05
## 4 140    4.08
## 5 140    4.10
## 6 140    4.13
```

`d_grid` contains every combination of `mu` and `sigma` across their specified values. Instead of base R `sapply()`, we'll do the computations by making a custom function which we'll plug into `purrr::map2()`.

```
grid_function <- function(mu, sigma){
  dnorm(d2$height, mean = mu, sd = sigma, log = T) %>%
    sum()
}
```

Now we're ready to complete the tibble.

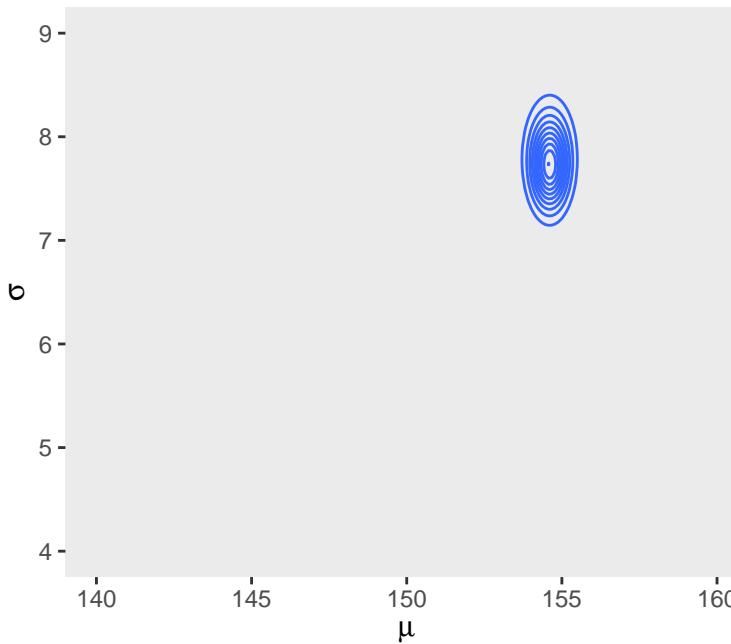
```
d_grid <-
d_grid %>%
  mutate(log_likelihood = map2(mu, sigma, grid_function)) %>%
  unnest() %>%
  mutate(prior_mu      = dnorm(mu,      mean = 178, sd   = 20, log = T),
         prior_sigma    = dunif(sigma, min  = 0,   max = 50, log = T)) %>%
  mutate(product       = log_likelihood + prior_mu + prior_sigma) %>%
  mutate(probability   = exp(product - max(product)))

head(d_grid)
```

```
## # A tibble: 6 x 7
##       mu   sigma log_likelihood prior_mu prior_sigma product probability
##   <dbl>  <dbl>        <dbl>     <dbl>     <dbl>     <dbl>
## 1   140     4       -3813.    -5.72    -3.91    -3822.      0
## 2   140    4.03     -3778.    -5.72    -3.91    -3787.      0
## 3   140    4.05     -3743.    -5.72    -3.91    -3753.      0
## 4   140    4.08     -3709.    -5.72    -3.91    -3719.      0
## 5   140    4.10     -3676.    -5.72    -3.91    -3686.      0
## 6   140    4.13     -3644.    -5.72    -3.91    -3653.      0
```

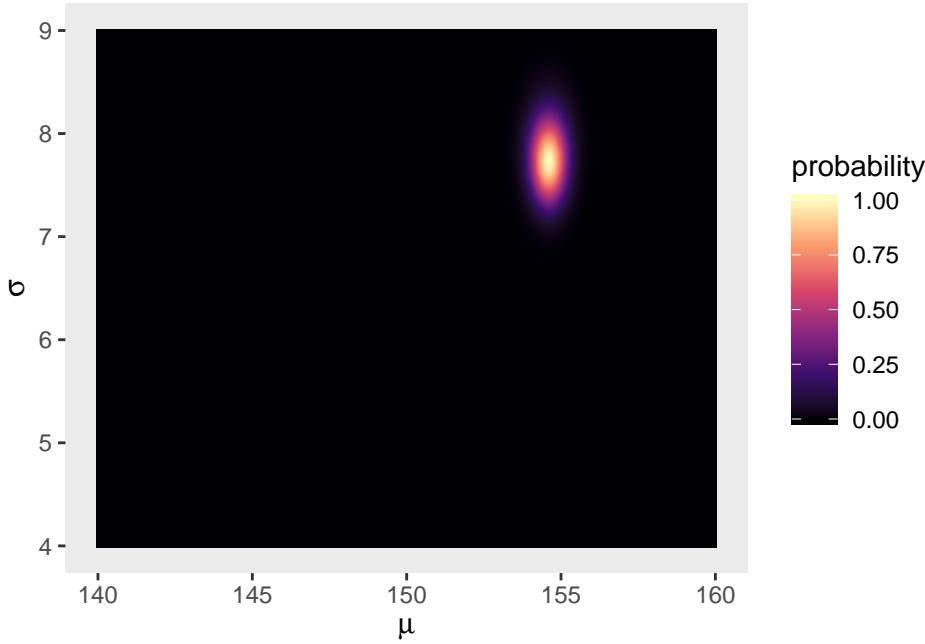
In the final `d_grid`, the `probability` vector contains the posterior probabilities across values of `mu` and `sigma`. We can make a contour plot with `geom_contour()`.

```
d_grid %>%
  ggplot(aes(x = mu, y = sigma, z = probability)) +
  geom_contour() +
  labs(x = expression(mu),
       y = expression(sigma)) +
  coord_cartesian(xlim = range(d_grid$mu),
                  ylim = range(d_grid$sigma)) +
  theme(panel.grid = element_blank())
```



We'll make our heat map with `geom_raster(aes(fill = probability))`.

```
d_grid %>%
  ggplot(aes(x = mu, y = sigma)) +
  geom_raster(aes(fill = probability),
              interpolate = T) +
  scale_fill_viridis_c(option = "A") +
  labs(x = expression(mu),
       y = expression(sigma)) +
  theme(panel.grid = element_blank())
```



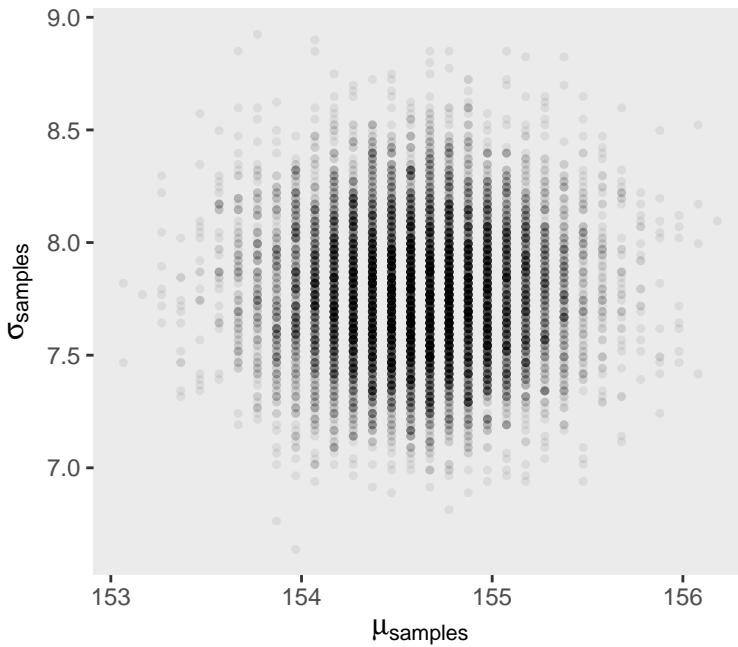
4.3.4 Sampling from the posterior.

We can use `dplyr::sample_n()` to sample rows, with replacement, from `d_grid`.

```
set.seed(4)
d_grid_samples <-
```

```
d_grid %>%
  sample_n(size = 1e4, replace = T, weight = probability)

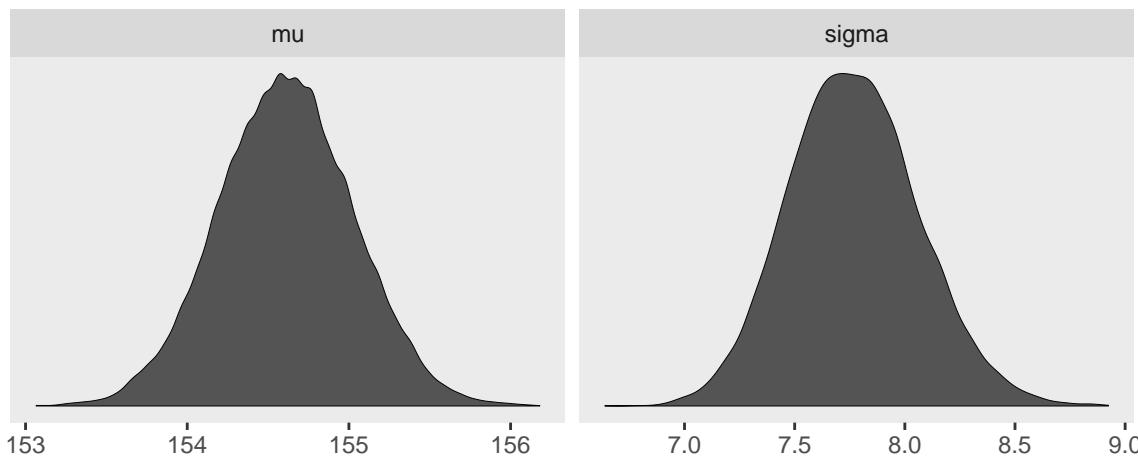
d_grid_samples %>%
  ggplot(aes(x = mu, y = sigma)) +
  geom_point(size = .9, alpha = 1/15) +
  scale_fill_viridis_c() +
  labs(x = expression(mu[samples]), y = expression(sigma[samples])) +
  theme(panel.grid = element_blank())
```



We can use `gather()` and then `facet_wrap()` to plot the densities for both `mu` and `sigma` at once.

```
d_grid_samples %>%
  select(mu, sigma) %>%
  gather() %>%

  ggplot(aes(x = value)) +
  geom_density(fill = "grey33", size = 0) +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab(NULL) +
  theme(panel.grid = element_blank()) +
  facet_wrap(~key, scales = "free")
```



We'll use the tidybayes package to compute their posterior modes and 95% HDIs.

```
library(tidybayes)

d_grid_samples %>%
  select(mu, sigma) %>%
  gather() %>%
  group_by(key) %>%
  mode_hdi(value)

## # A tibble: 2 x 7
##   key     value .lower .upper .width .point .interval
##   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <chr>  <chr>
## 1 mu      155.   154.   155.    0.95 mode    hdi
## 2 sigma    7.71   7.19   8.32    0.95 mode    hdi
```

Let's say you wanted their posterior medians and 50% quantile-based intervals, instead. Just switch out the last line for `median_qi(value, .width = .5)`.

4.3.4.1 Overthinking: Sample size and the normality of σ 's posterior.

Since we'll be fitting models with brms almost exclusively from here on out, this section is largely mute. But we'll do it anyway for the sake of practice. I'm going to break the steps up like before rather than compress the code together. Here's d3.

```
set.seed(4)
(d3 <- sample(d2$height, size = 20))
```

```
## [1] 149.225 156.845 150.495 149.225 158.750 169.545 167.005 160.655 141.605 146.050 168.910 163.195
## [13] 171.450 165.100 148.590 156.210 160.655 149.225 147.320 152.400
```

For our first step using d3, we'll redefine `d_grid`.

```
n <- 200

# note we've redefined the ranges of `mu` and `sigma`
d_grid <-
  tibble(mu      = seq(from = 150, to = 170, length.out = n),
         sigma   = seq(from = 4,   to = 20,   length.out = n)) %>%
  expand(mu, sigma)
```

Second, we'll redefine our custom `grid_function()` function to operate over the `height` values of d3.

```
grid_function <- function(mu, sigma){
  dnorm(d3, mean = mu, sd = sigma, log = T) %>%
  sum()
}
```

Now we'll use the amended `grid_function()` to make the posterior.

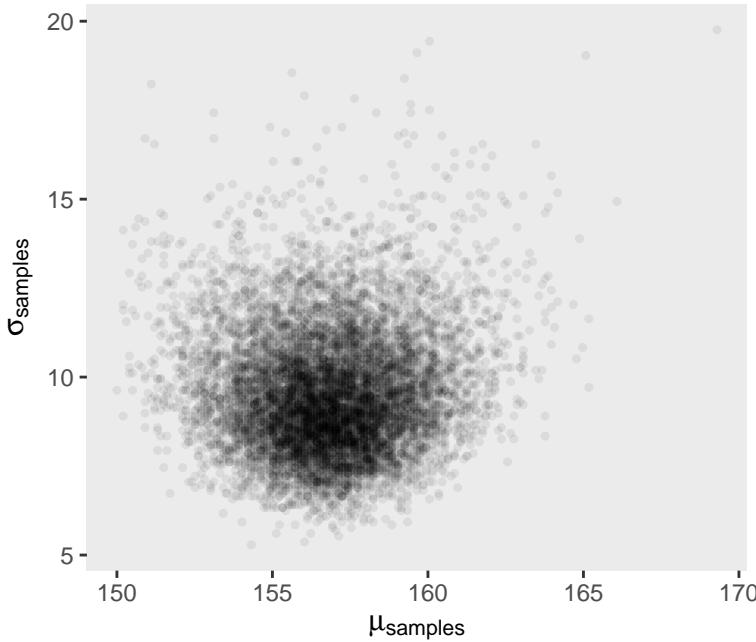
```
d_grid <-
  d_grid %>%
  mutate(log_likelihood = map2_dbl(mu, sigma, grid_function)) %>%
  mutate(prior_mu       = dnorm(mu,      mean = 178, sd = 20, log = T),
         prior_sigma    = dunif(sigma, min = 0, max = 50, log = T)) %>%
  mutate(product       = log_likelihood + prior_mu + prior_sigma) %>%
  mutate(probability   = exp(product - max(product)))
```

Did you catch our use of `purrr::map2_dbl()`, there, in place of `purrr::map2()`? It turns out that `purrr::map()` and `purrr::map2()` always return a list (see [here](#) and [here](#)). But as [Phil Straforelli](#) kindly pointed out, we can add the `_dbl` suffix to those functions, which will instruct the `purrr` package to return a double vector (i.e., a [common kind of numeric vector](#)). The advantage of that approach is we no longer need to follow our `map()` or `map2()` lines with `unnest()`. To learn more about the ins and outs of the `map()` family, check out [this section](#) from *R4DS* or Jenny Bryan's [purrr tutorial](#).

Next we'll `sample_n()` and plot.

```
set.seed(4)
d_grid_samples <-
  d_grid %>%
  sample_n(size = 1e4, replace = T, weight = probability)

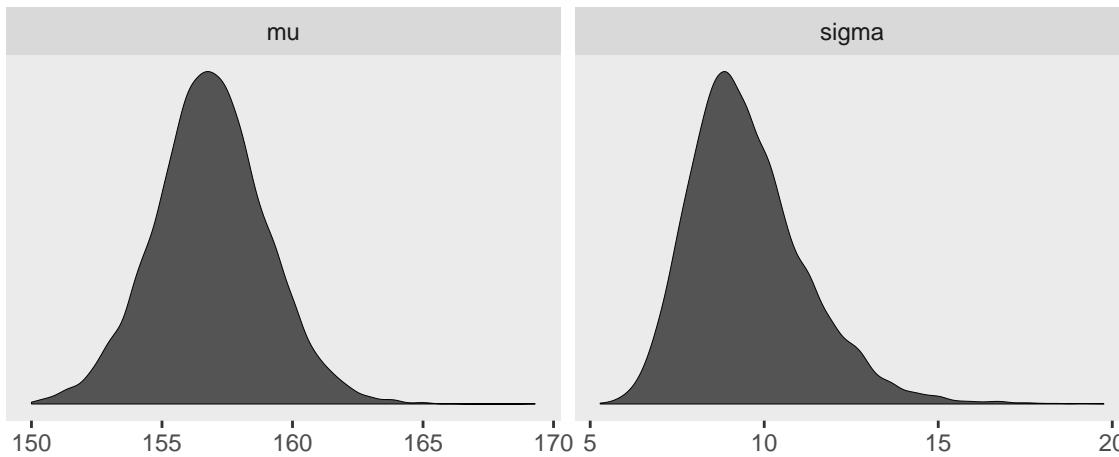
d_grid_samples %>%
  ggplot(aes(x = mu, y = sigma)) +
  geom_point(size = .9, alpha = 1/15) +
  scale_fill_viridis_c() +
  labs(x = expression(mu[samples]),
       y = expression(sigma[samples])) +
  theme(panel.grid = element_blank())
```



Behold the updated densities.

```
d_grid_samples %>%
  select(mu, sigma) %>%
  gather() %>%

  ggplot(aes(x = value)) +
  geom_density(fill = "grey33", size = 0) +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab(NULL) +
  theme(panel.grid = element_blank()) +
  facet_wrap(~key, scales = "free")
```



Sigma's not so Gaussian with that small n .

4.3.5 Fitting the model with `map()` `brm()`.

We won't actually use `rethinking::map()`—which you should not conflate with `purrr::map()`—but will jump straight to the primary brms modeling function, `brm()`. In the text, McElreath indexed his models with names like `m4.1`. I will largely follow that convention, but will replace the m with a b to stand for the brms package. Here's the first model.

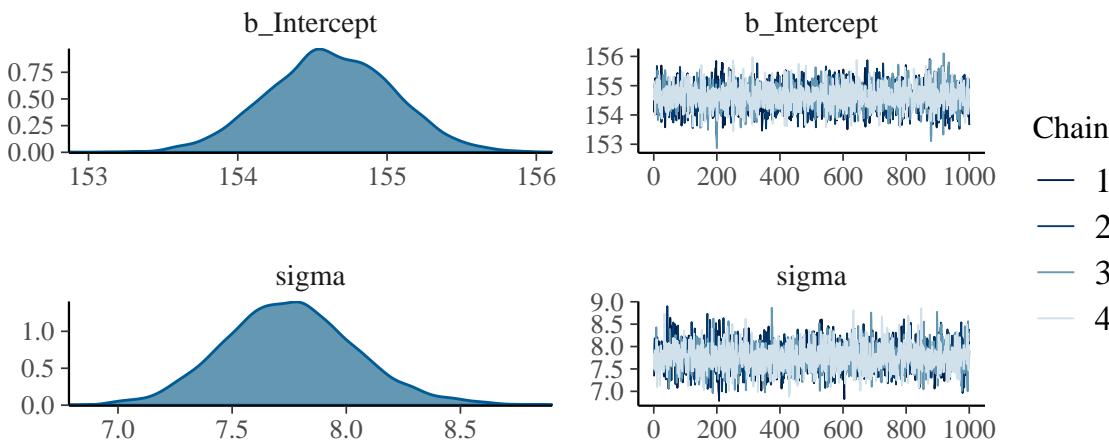
```
b4.1 <-
  brm(data = d2, family = gaussian,
       height ~ 1,
       prior = c(prior(normal(178, 20), class = Intercept),
                 prior(uniform(0, 50), class = sigma)),
       iter = 31000, warmup = 30000, chains = 4, cores = 4,
       seed = 4)
```

McElreath's uniform prior for σ was rough on brms. It took an unusually-large number of warmup iterations before the chains sampled properly. As McElreath covered in Chapter 8, HMC tends to work better when you default to a half Cauchy for σ . Here's how to do so.

```
b4.1_half_cauchy <-
  brm(data = d2, family = gaussian,
       height ~ 1,
       prior = c(prior(normal(178, 20), class = Intercept),
                 prior(cauchy(0, 1), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       seed = 4)
```

This leads to an important point. After running model with Hamiltonian Monte Carlo (HMC), it's a good idea to inspect the chains. As we'll see, McElreath covered this in Chapter 8. Here's a typical way to do so in brms.

```
plot(b4.1_half_cauchy)
```



If you want detailed diagnostics for the HMC chains, call `launch_shinystan(b4.1)`. That'll keep you busy for a while. But anyway, the chains look good. We can reasonably trust the results.

Here's how to get the model summary of our `brm()` object.

```
print(b4.1_half_cauchy)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 1
##   Data: d2 (Number of observations: 352)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept    154.62     0.42    153.81    155.43      2725 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       7.75     0.29     7.21     8.36      3317 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

The `summary()` function works in a similar way.

You can also get a [Stan-like summary](#) with this:

```
b4.1_half_cauchy$fit
```

```
## Inference for Stan model: 7677f26c8551f442ed3bf486232e7b04.
## 4 chains, each with iter=2000; warmup=1000; thin=1;
## post-warmup draws per chain=1000, total post-warmup draws=4000.
##
##           mean se_mean    sd    2.5%    25%    50%    75%   97.5% n_eff Rhat
## b_Intercept 154.62    0.01 0.42  153.81  154.34  154.62  154.91  155.43  2725    1
## sigma       7.75    0.01 0.29    7.21    7.56    7.75    7.94    8.36  3317    1
## lp__      -1227.52   0.03 1.01 -1230.29 -1227.90 -1227.21 -1226.80 -1226.54  1317    1
##
## Samples were drawn using NUTS(diag_e) at Sat Apr 20 00:47:57 2019.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Whereas rethinking defaults to 89% intervals, using `print()` or `summary()` with `brms` models defaults to 95% intervals. Unless otherwise specified, I will stick with 95% intervals throughout. However, if you really want those 89% intervals, an easy way is with the `prob` argument within `brms::summary()` or `brms::print()`.

```
summary(b4.1_half_cauchy, prob = .89)
```

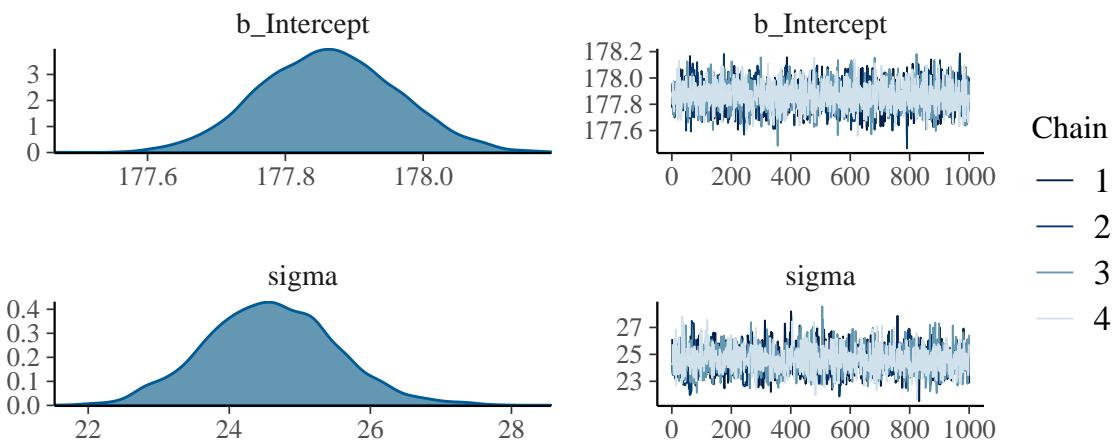
```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 1
##   Data: d2 (Number of observations: 352)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-89% CI u-89% CI Eff.Sample Rhat
## Intercept     154.62      0.42    153.96    155.28        2725 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-89% CI u-89% CI Eff.Sample Rhat
## sigma       7.75      0.29      7.30      8.24        3317 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Anyways, here's the shockingly-narrow- μ -prior model.

```
b4.2 <-
  brm(data = d2, family = gaussian,
    height ~ 1,
    prior = c(prior(normal(178, .1), class = Intercept),
              prior(uniform(0, 50), class = sigma)),
    iter = 3000, warmup = 2000, chains = 4, cores = 4,
    seed = 4)
```

Check the chains.

```
plot(b4.2)
```



I had to increase the `warmup` due to convergence issues. After doing so, everything looks to be on the up and up. The chains look great. And again, to learn more about these technical details, check out Chapter 8.

Here's the model `summary()`.

```
summary(b4.2)
```

```
## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: height ~ 1
## Data: d2 (Number of observations: 352)
## Samples: 4 chains, each with iter = 3000; warmup = 2000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept    177.86     0.10   177.67   178.06       3434 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma      24.59     0.92   22.82    26.47       1735 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

4.3.6 Sampling from a `map()` `brm()` fit.

`brms` doesn't seem to have a convenience function that works the way `vcov()` does for `rethinking`. For example:

```
vcov(b4.1_half_cauchy)
```

```
##             Intercept
## Intercept 0.1733533
```

This only returns the first element in the matrix it did for `rethinking`. That is, it appears `brms::vcov()` only returns the variance/covariance matrix for the single-level β parameters (i.e., those used to model μ).

However, if you really wanted this information, you could get it after putting the HMC chains in a data frame.

```
post <- posterior_samples(b4.1_half_cauchy)

head(post)
```

```
##   b_Intercept     sigma     lp__
## 1   155.0785 7.335334 -1228.205
## 2   154.0975 7.652924 -1227.324
## 3   154.8050 7.499036 -1226.946
## 4   155.2052 7.574270 -1227.748
## 5   155.2581 7.565656 -1227.970
## 6   154.0118 7.612666 -1227.663
```

Now `select()` the columns containing the draws from the desired parameters and feed them into `cov()`.

```
select(post, b_Intercept:sigma) %>%
  cov()
```

```
##             b_Intercept     sigma
## b_Intercept 0.173353306 -0.003242316
## sigma        -0.003242316  0.084394323
```

That was “(1) a vector of variances for the parameters and (2) a correlation matrix” for them (p. 90). Here are just the variances (i.e., the diagonal elements) and the correlation matrix.

```
# variances
select(post, b_Intercept:sigma) %>%
  cov() %>%
  diag()

## b_Intercept      sigma
##  0.17335331  0.08439432

# correlation
post %>%
  select(b_Intercept, sigma) %>%
  cor()

##           b_Intercept      sigma
## b_Intercept  1.00000000 -0.02680604
## sigma        -0.02680604  1.00000000
```

With our `post <- posterior_samples(b4.1_half_cauchy)` code from a few lines above, we've already done the brms version of what McElreath did with `extract.samples()` on page 90. However, what happened under the hood was different. Whereas rethinking used the `mvnrm()` function from the [MASS package](#), in brms we just extracted the iterations of the HMC chains and put them in a data frame.

```
str(post)

## 'data.frame': 4000 obs. of 3 variables:
## $ b_Intercept: num 155 154 155 155 155 ...
## $ sigma       : num 7.34 7.65 7.5 7.57 7.57 ...
## $ lp__        : num -1228 -1227 -1227 -1228 -1228 ...
```

Notice how our data frame, `post`, includes a third vector, `lp__`. That's the log posterior. See the [brms reference manual](#) or the “The Log-Posterior (function and gradient)” section of the Stan Development Team's [RStan: the R interface to Stan](#) for details. The log posterior will largely be outside of our focus in this project.

The `summary()` function doesn't work for brms posterior data frames quite the way `precis()` does for posterior data frames from the rethinking package. E.g.,

```
summary(post[, 1:2])

##   b_Intercept      sigma
## Min.    :152.9  Min.   :6.785
## 1st Qu.:154.3  1st Qu.:7.561
## Median :154.6  Median :7.748
## Mean   :154.6  Mean   :7.754
## 3rd Qu.:154.9  3rd Qu.:7.938
## Max.   :156.1  Max.   :8.902
```

Here's one option using the transpose of a `quantile()` call nested within `apply()`, which is a very general function you can learn more about [here](#) or [here](#).

```
t(apply(post[, 1:2], 2, quantile, probs = c(.5, .025, .75)))

##           50%      2.5%      75%
## b_Intercept 154.619016 153.814860 154.909843
## sigma        7.748051  7.210529  7.938047
```

The base R code is compact, but somewhat opaque. Here's how to do something similar with more explicit tidyverse code.

```
post %>%
  select(-lp__)
  gather(parameter) %>%
  group_by(parameter) %>%
  summarise(mean = mean(value),
            SD = sd(value),
            `2.5_percentile` = quantile(value, probs = .025),
            `97.5_percentile` = quantile(value, probs = .975)) %>%
  mutate_if(is.numeric, round, digits = 2)
```

```
## # A tibble: 2 x 5
##   parameter      mean     SD `2.5_percentile` `97.5_percentile`
##   <chr>        <dbl>    <dbl>       <dbl>           <dbl>
## 1 b_Intercept 155.     0.42      154.          155.
## 2 sigma        7.75    0.290     7.21          8.36
```

You can always get pretty similar information by just putting the `brm()` fit object into `posterior_summary()`.

```
posterior_summary(b4.1_half_cauchy)
```

```
##               Estimate Est.Error      Q2.5      Q97.5
## b_Intercept 154.623104 0.4163572 153.814860 155.433461
## sigma        7.754454 0.2905070  7.210529  8.361077
## lp__        -1227.519041 1.0103147 -1230.287040 -1226.540850
```

And if you're willing to drop the posterior *SDs*, you can use `tidybayes::mean_qi()`, too.

```
post %>%
  select(-lp__)
  gather(parameter) %>%
  group_by(parameter) %>%
  mean_qi(value)

## # A tibble: 2 x 7
##   parameter      value .lower .upper .width .point .interval
##   <chr>        <dbl>   <dbl>   <dbl>   <dbl>   <chr>   <chr>
## 1 b_Intercept 155.    154.    155.    0.95  mean    qi
## 2 sigma        7.75    7.21    8.36    0.95  mean    qi
```

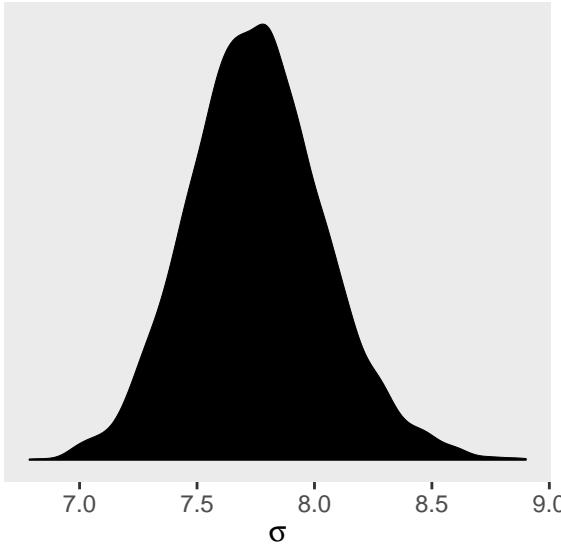
4.3.6.1 Overthinking: Under the hood with multivariate sampling.

Again, `brms::posterior_samples()` is not the same as `rethinking::extract.samples()`. Rather than use the `MASS::mvnrm()`, brms takes the iterations from the HMC chains. McElreath covered all of this in Chapter 8. You might also look at the brms [reference manual](#) or [GitHub page](#) for details.

4.3.6.2 Overthinking: Getting σ right.

There's no need to fret about this in brms. With HMC, we are not constraining the posteriors to the multivariate normal distribution. Here's our posterior density for σ .

```
ggplot(data = post,
       aes(x = sigma)) +
  geom_density(size = 1/10, fill = "black") +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab(expression(sigma)) +
  theme(panel.grid = element_blank())
```



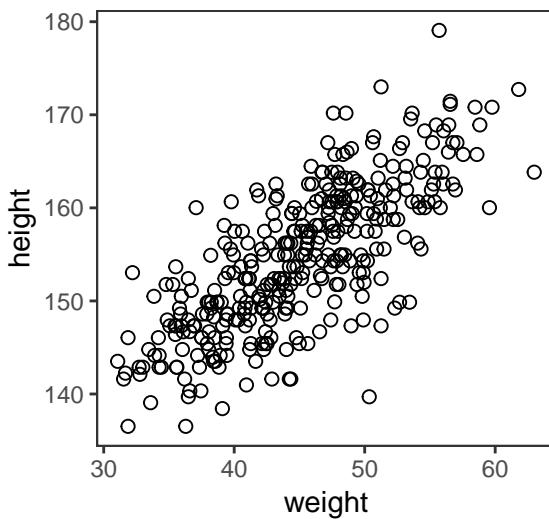
See? HMC handled the mild skew just fine.

But sometimes you want to actually model σ , such as in the case where your variances are systematically heterogeneous. Bürkner calls these kinds of models distributional models, which you can learn more about in his vignette [Estimating Distributional Models with brms](#). As he explained in the vignette, you actually model $\log(\sigma)$ in those instances. If you're curious, we'll practice with a model like this in Chapter 9.

4.4 Adding a predictor

Here's our scatter plot of `weight` and `height`.

```
ggplot(data = d2,
       aes(x = weight, y = height)) +
  geom_point(shape = 1, size = 2) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



4.4.1 The linear model strategy

In our new univariable model

$$\begin{aligned}
 h_i &\sim \text{Normal}(\mu_i, \sigma) \\
 \mu_i &= \alpha + \beta x_i \\
 \alpha &\sim \text{Normal}(178, 100) \\
 \beta &\sim \text{Normal}(0, 10) \\
 \sigma &\sim \text{Uniform}(0, 50)
 \end{aligned}$$

4.4.2 Fitting the model.

The `brms::brm()` syntax doesn't mirror the statistical notation. But here are the analogues to the exposition at the bottom of page 95.

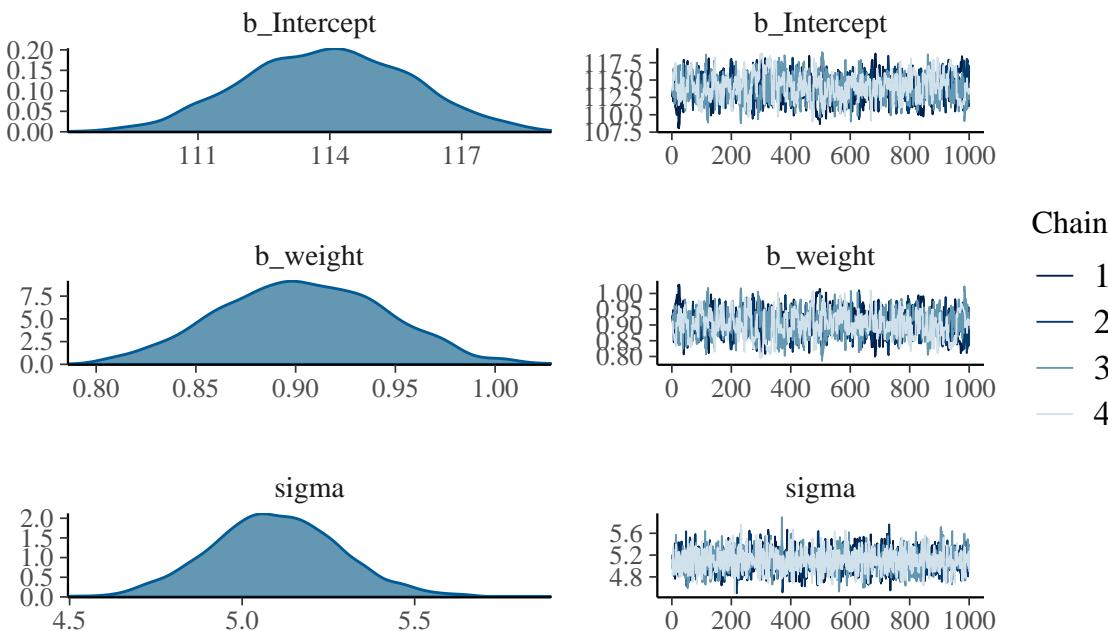
- $h_i \sim \text{Normal}(\mu_i, \sigma)$: `family = gaussian`
- $\mu_i = \alpha + \beta x_i$: `height ~ 1 + weight`
- $\alpha \sim \text{Normal}(156, 100)$: `prior(normal(156, 100), class = Intercept)`
- $\beta \sim \text{Normal}(0, 10)$: `prior(normal(0, 10), class = b)`
- $\sigma \sim \text{Uniform}(0, 50)$: `prior(uniform(0, 50), class = sigma)`

Thus, to add a predictor you just the `+` operator in the model formula.

```
b4.3 <-  
  brm(data = d2, family = gaussian,  
        height ~ 1 + weight,  
        prior = c(prior(normal(156, 100), class = Intercept),  
                  prior(normal(0, 10), class = b),  
                  prior(uniform(0, 50), class = sigma)),  
        iter = 41000, warmup = 40000, chains = 4, cores = 4,  
        seed = 4)
```

This was another example of how using a uniform prior for σ required we use an unusually large number of `warmup` iterations before the HMC chains converged on the posterior. Change the prior to `cauchy(0, 1)` and the chains converge with no problem, resulting in much better effective samples, too. Here are the trace plots.

```
plot(b4.3)
```



4.4.3 Interpreting the model fit.

“One trouble with statistical models is that they are hard to understand” (p. 97). Welcome to the world of applied statistics.

4.4.3.1 Tables of estimates.

With a little [] subsetting we can exclude the log posterior from the summary.

```
posterior_summary(b4.3)[1:3, ]
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	113.938784	1.8713415	110.3740463	117.5393583
## b_weight	0.903689	0.0412386	0.8244307	0.9816048
## sigma	5.096834	0.1874516	4.7311309	5.4740004

Again, brms doesn’t have a convenient `corr = TRUE` argument for `plot()` or `summary()`. But you can get that information after putting the chains in a data frame.

```
posterior_samples(b4.3) %>%
  select(-lp__)
  cor() %>%
  round(digits = 2)
```

	b_Intercept	b_weight	sigma
## b_Intercept	1.00	-0.99	-0.01
## b_weight	-0.99	1.00	0.01
## sigma	-0.01	0.01	1.00

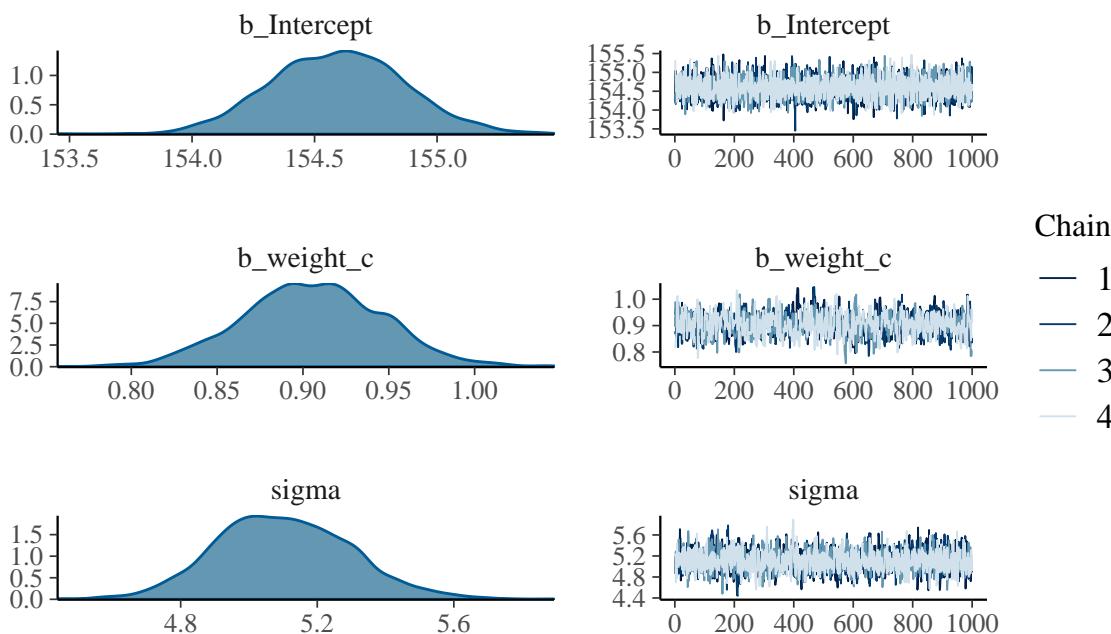
With centering, we can reduce the correlations among the parameters.

```
d2 <-
d2 %>%
  mutate(weight_c = weight - mean(weight))
```

Fit the `weight_c` model, `b4.4`.

```
b4.4 <-
  brm(data = d2, family = gaussian,
    height ~ 1 + weight_c,
    prior = c(prior(normal(178, 100), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(uniform(0, 50), class = sigma)),
    iter = 46000, warmup = 45000, chains = 4, cores = 4,
    seed = 4)
```

```
plot(b4.4)
```



```
posterior_summary(b4.4)[1:3, ]
```

```
##           Estimate   Est.Error      Q2.5     Q97.5
## b_Intercept 154.5972099 0.27167270 154.0732619 155.1449945
## b_weight_c    0.9056644 0.04074593  0.8271512  0.9841592
## sigma         5.1015725 0.19645580  4.7371764  5.5066741
```

Like before, the uniform prior required extensive `warmup` iterations to produce a good posterior. This is easily fixed using a half Cauchy prior, instead. Anyways, the effective samples improved. Here's the parameter correlation info.

```
posterior_samples(b4.4) %>%
  select(-lp_) %>%
  cor() %>%
  round(digits = 2)
```

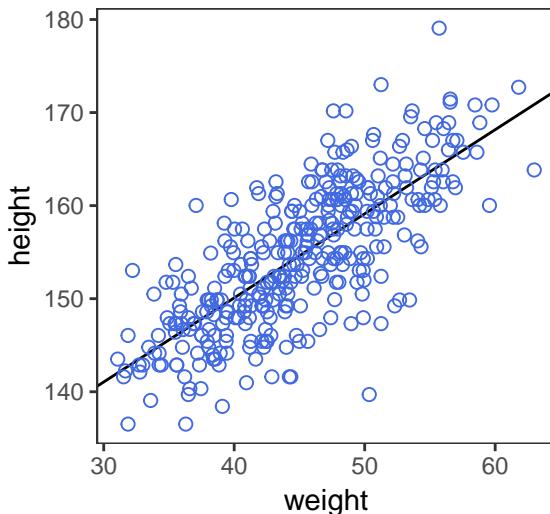
```
##           b_Intercept b_weight_c sigma
## b_Intercept     1.00     0.02 -0.01
## b_weight_c      0.02     1.00  0.04
## sigma          -0.01     0.04  1.00
```

See? Now all the correlations are quite low. Also, if you prefer a visual approach, you might do `pairs(b4.4)`.

4.4.3.2 Plotting posterior inference against the data.

Here is the code for Figure 4.4. Note our use of the `fixef()` function.

```
d2 %>%
  ggplot(aes(x = weight, y = height)) +
  geom_abline(intercept = fixef(b4.3)[1],
              slope    = fixef(b4.3)[2]) +
  geom_point(shape = 1, size = 2, color = "royalblue") +
  theme_bw() +
  theme(panel.grid = element_blank())
```



In the brms reference manual, Bürkner described the job of the `fixef()` function as “extract[ing] the population-level (‘fixed’) effects from a `brmsfit` object”. If you’re new to multilevel models, it might not be clear what he meant by “population-level” or “fixed” effects. Don’t worry. That’ll all become clear starting around Chapter 12. In the meantime, just think of them as the typical regression parameters, minus σ .

4.4.3.3 Adding uncertainty around the mean.

Be default, we extract all the posterior iterations with `posterior_samples()`.

```
post <- posterior_samples(b4.3)

post %>%
  slice(1:5) # this serves a similar function as `head()``
```

	b_Intercept	b_weight	sigma	lp__
## 1	113.1684	0.9199020	5.151709	-1082.192
## 2	113.0394	0.9195661	5.181224	-1082.471
## 3	113.5766	0.9200660	5.196852	-1083.202
## 4	112.3897	0.9475964	5.101521	-1083.790
## 5	111.8950	0.9400941	5.096093	-1083.521

Here are the four models leading up to McElreath’s Figure 4.5. To reduce my computation time, I used a half Cauchy($0, 1$) prior on σ . If you are willing to wait for the warmups, switching that out for McElreath’s uniform prior should work fine as well.

```
n <- 10

b.10 <-
  brm(data = d2 %>%
    slice(1:n), # note our tricky use of `n` and `slice()`
    family = gaussian,
    height ~ 1 + weight,
    prior = c(prior(normal(178, 100), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 1), class = sigma)),
    iter = 2000, warmup = 1000, chains = 4, cores = 4,
    seed = 4)

n <- 50

b.50 <-
  brm(data = d2 %>%
```

```

slice(1:n),
family = gaussian,
height ~ 1 + weight,
prior = c(prior(normal(178, 100), class = Intercept),
          prior(normal(0, 10), class = b),
          prior(cauchy(0, 1), class = sigma)),
iter = 2000, warmup = 1000, chains = 4, cores = 4,
seed = 4)

n <- 150

b.150 <-
  brm(data = d2 %>%
    slice(1:n),
    family = gaussian,
    height ~ 1 + weight,
    prior = c(prior(normal(178, 100), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 1), class = sigma)),
    iter = 2000, warmup = 1000, chains = 4, cores = 4,
    seed = 4)

n <- 352

b.352 <-
  brm(data = d2 %>%
    slice(1:n),
    family = gaussian,
    height ~ 1 + weight,
    prior = c(prior(normal(178, 100), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 1), class = sigma)),
    iter = 2000, warmup = 1000, chains = 4, cores = 4,
    seed = 4)

```

I'm not going to clutter up the document with all the trace plots and coefficient summaries from these four models. But here's how to get that information.

```

plot(b.10)
print(b.10)

plot(b.50)
print(b.50)

plot(b.150)
print(b.150)

plot(b.352)
print(b.352)

```

We'll need to put the chains of each model into data frames.

```

post10 <- posterior_samples(b.10)
post50 <- posterior_samples(b.50)
post150 <- posterior_samples(b.150)
post352 <- posterior_samples(b.352)

```

Here is the code for the four individual plots.

```

p10 <-
  ggplot(data = d2[1:10 , ],
         aes(x = weight, y = height)) +
  geom_abline(intercept = post10[1:20, 1],
              slope     = post10[1:20, 2],
              size = 1/3, alpha = .3) +
  geom_point(shape = 1, size = 2, color = "royalblue") +
  coord_cartesian(xlim = range(d2$weight),
                  ylim = range(d2$height)) +
  labs(subtitle = "N = 10") +
  theme_bw() +
  theme(panel.grid = element_blank())

p50 <-
  ggplot(data = d2[1:50 , ],
         aes(x = weight, y = height)) +
  geom_abline(intercept = post50[1:20, 1],
              slope     = post50[1:20, 2],
              size = 1/3, alpha = .3) +
  geom_point(shape = 1, size = 2, color = "royalblue") +
  coord_cartesian(xlim = range(d2$weight),
                  ylim = range(d2$height)) +
  labs(subtitle = "N = 50") +
  theme_bw() +
  theme(panel.grid = element_blank())

p150 <-
  ggplot(data = d2[1:150 , ],
         aes(x = weight, y = height)) +
  geom_abline(intercept = post150[1:20, 1],
              slope     = post150[1:20, 2],
              size = 1/3, alpha = .3) +
  geom_point(shape = 1, size = 2, color = "royalblue") +
  coord_cartesian(xlim = range(d2$weight),
                  ylim = range(d2$height)) +
  labs(subtitle = "N = 150") +
  theme_bw() +
  theme(panel.grid = element_blank())

p352 <-
  ggplot(data = d2[1:352 , ],
         aes(x = weight, y = height)) +
  geom_abline(intercept = post352[1:20, 1],
              slope     = post352[1:20, 2],
              size = 1/3, alpha = .3) +
  geom_point(shape = 1, size = 2, color = "royalblue") +
  coord_cartesian(xlim = range(d2$weight),
                  ylim = range(d2$height)) +
  labs(subtitle = "N = 352") +
  theme_bw() +
  theme(panel.grid = element_blank())

```

Note how we used the good old bracket syntax (e.g., `d2[1:10 ,]`) to index rows from our `d2` data. With tidyverse-style syntax, we could have done `slice(d2, 1:10)` or `d2 %>% slice(1:10)` instead.

Anyway, we saved each of these plots as objects. With a little help of the `multiplot()` function we are going to arrange those plot objects into a grid in order to reproduce Figure 4.5.

Behold the code for the `multiplot()` function:

```

multiplot <- function(..., plotlist=NULL, file, cols=1, layout=NULL) {
  library(grid)

  # Make a list from the ... arguments and plotlist
  plots <- c(list(...), plotlist)

  numPlots = length(plots)

  # If layout is NULL, then use 'cols' to determine layout
  if (is.null(layout)) {
    # Make the panel
    # ncol: Number of columns of plots
    # nrow: Number of rows needed, calculated from # of cols
    layout <- matrix(seq(1, cols * ceiling(numPlots/cols)),
                    ncol = cols, nrow = ceiling(numPlots/cols))
  }

  if (numPlots==1) {
    print(plots[[1]])

  } else {
    # Set up the page
    grid.newpage()
    pushViewport(viewport(layout = grid.layout(nrow(layout), ncol(layout))))

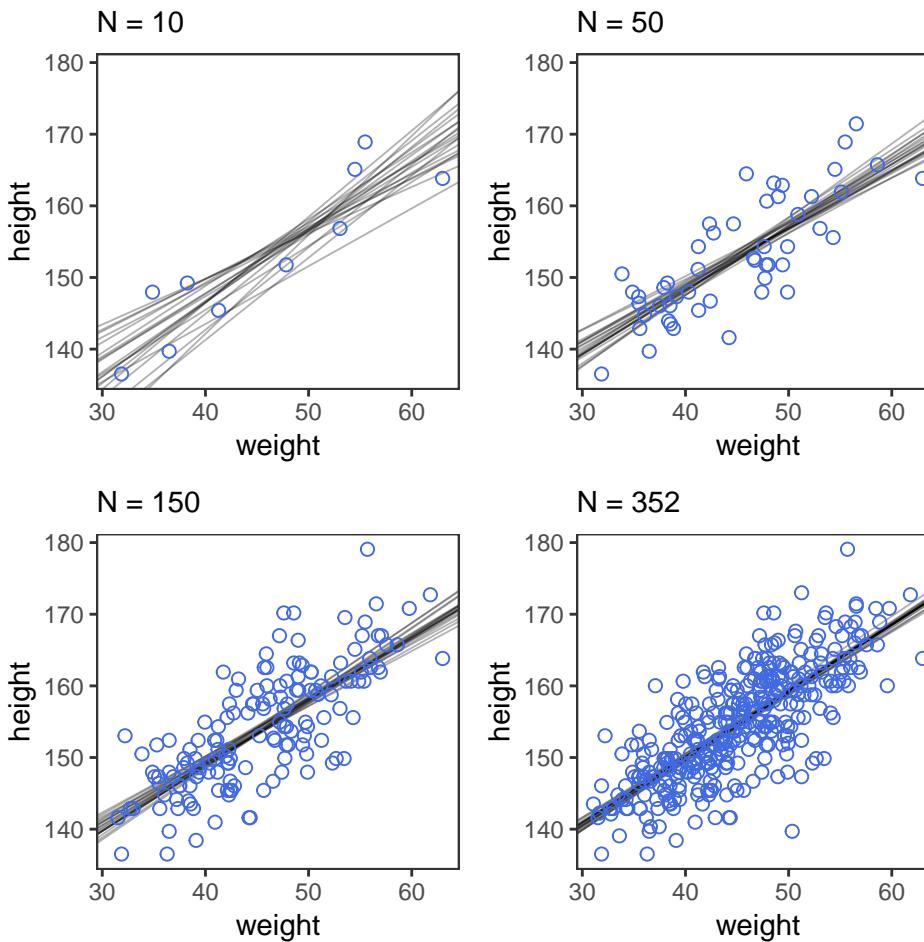
    # Make each plot, in the correct location
    for (i in 1:numPlots) {
      # Get the i,j matrix positions of the regions that contain this subplot
      matchidx <- as.data.frame(which(layout == i, arr.ind = TRUE))

      print(plots[[i]], vp = viewport(layout.pos.row = matchidx$row,
                                      layout.pos.col = matchidx$col))
    }
  }
}

```

We're finally ready to use `multiplot()` to make Figure 4.5.

```
multiplot(p10, p150, p50, p352, cols = 2)
```



4.4.3.4 Plotting regression intervals and contours.

Remember, if you want to plot McElreath's `mu_at_50` with `ggplot2`, you'll need to save it as a data frame or a tibble.

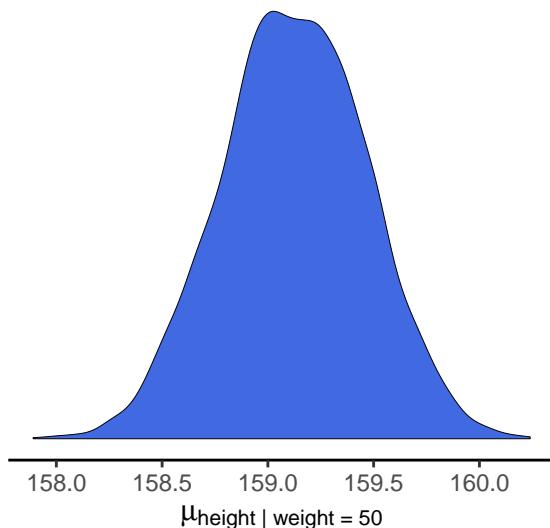
```
mu_at_50 <-
  post %>%
  transmute(mu_at_50 = b_Intercept + b_weight * 50)

head(mu_at_50)
```

```
##   mu_at_50
## 1 159.1635
## 2 159.0178
## 3 159.5799
## 4 159.7695
## 5 158.8997
## 6 158.9811
```

And here is a version McElreath's Figure 4.6 density plot.

```
mu_at_50 %>%
  ggplot(aes(x = mu_at_50)) +
  geom_density(size = 0, fill = "royalblue") +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(x = expression(mu["height | weight = 50"])) +
  theme_classic()
```



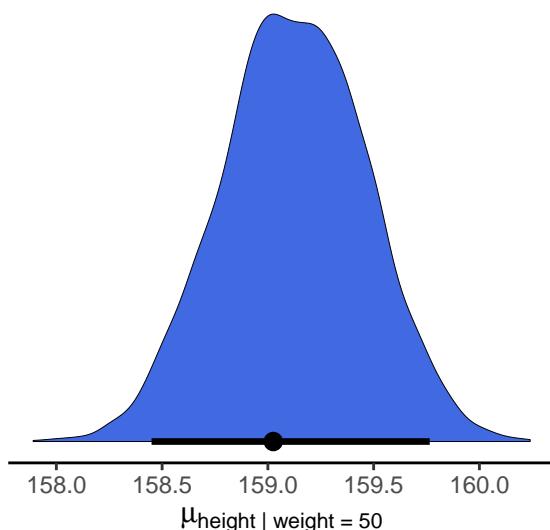
We'll use `mean_hdi()` to get both 89% and 95% HPDIs along with the mean.

```
mean_hdi(mu_at_50[, 1], .width = c(.89, .95))

##           y      ymin      ymax .width .point .interval
## 1 159.1232 158.5914 159.6856   0.89   mean     hdi
## 2 159.1232 158.4501 159.7652   0.95   mean     hdi
```

If you wanted to express those sweet 95% HPDIs on your density plot, you might use `tidybayes::stat_pointintervalh()`. Since `stat_pointintervalh()` also returns a point estimate, we'll throw in the mode.

```
mu_at_50 %>%
  ggplot(aes(x = mu_at_50)) +
  geom_density(size = 0, fill = "royalblue") +
  stat_pointintervalh(aes(y = 0),
                      point_interval = mode_hdi, .width = .95) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(x = expression(mu["height \mid weight = 50"])) +
  theme_classic()
```



In brms, you would use `fitted()` to do what McElreath accomplished with `link()`.

```
mu <- fitted(b4.3, summary = F)

str(mu)
```

```
## num [1:4000, 1:352] 157 157 158 158 157 ...
```

When you specify `summary = F`, `fitted()` returns a matrix of values with as many rows as there were post-warmup iterations across your HMC chains and as many columns as there were cases in your data. Because we had 4000 post-warmup iterations and $n = 352$, `fitted()` returned a matrix of 4000 rows and 352 vectors. If you omitted the `summary = F` argument, the default is `TRUE` and `fitted()` will return summary information instead.

Much like rethinking's `link()`, `fitted()` can accommodate custom predictor values with its `newdata` argument.

```
weight_seq <- tibble(weight = seq(from = 25, to = 70, by = 1))

mu <-
  fitted(b4.3,
    summary = F,
    newdata = weight_seq) %>%
  as_tibble() %>%
  # here we name the columns after the `weight` values from which they were computed
  set_names(25:70) %>%
  mutate(iter = 1:n())

str(mu)
```

Anticipating ggplot2, we went ahead and converted the output to a tibble. But we might do a little more data processing with the aid of `tidyverse::gather()`. With the `gather()` function, we'll convert the data from the wide format to the long format. If you're new to the distinction between wide and long data, you can learn more [here](#) or [here](#).

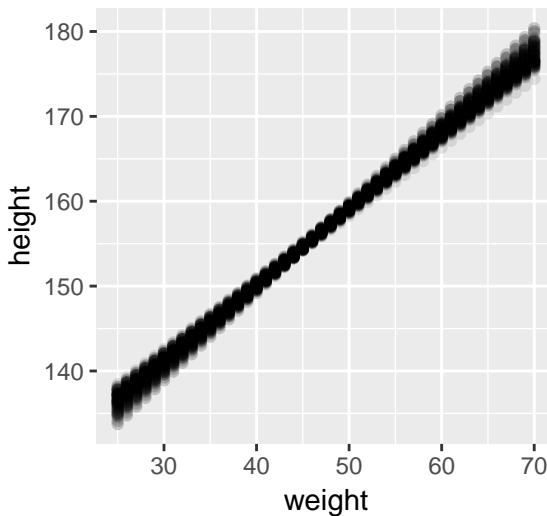
```
mu <-
  mu %>%
  gather(weight, height, -iter) %>%
  # We might reformat `weight` to numerals
  mutate(weight = as.numeric(weight))

head(mu)
```

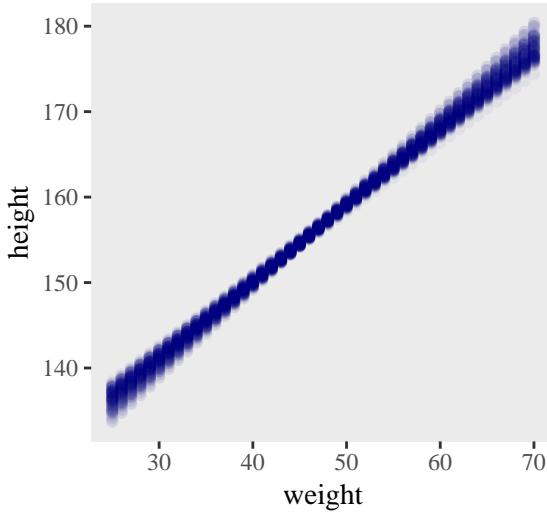
```
## # A tibble: 6 x 3
##   iter weight height
##   <int>  <dbl>  <dbl>
## 1     1     25    136.
## 2     2     25    136.
## 3     3     25    137.
## 4     4     25    136.
## 5     5     25    135.
## 6     6     25    138.
```

Enough data processing. Here we reproduce McElreath's Figure 4.7.a.

```
d2 %>%
  ggplot(aes(x = weight, y = height)) +
  geom_point(data = mu %>% filter(iter < 101),
             alpha = .1)
```



```
# or prettied up a bit
d2 %>%
  ggplot(aes(x = weight, y = height)) +
  geom_point(data = mu %>% filter(iter < 101),
             color = "navyblue", alpha = .05) +
  theme(text = element_text(family = "Times"),
        panel.grid = element_blank())
```



With `fitted()`, it's quite easy to plot a regression line and its intervals. Just omit the `summary = T` argument.

```
mu_summary <-
  fitted(b4.3,
         newdata = weight_seq) %>%
  as_tibble() %>%
  # let's tack on the `weight` values from `weight_seq`
  bind_cols(weight_seq)

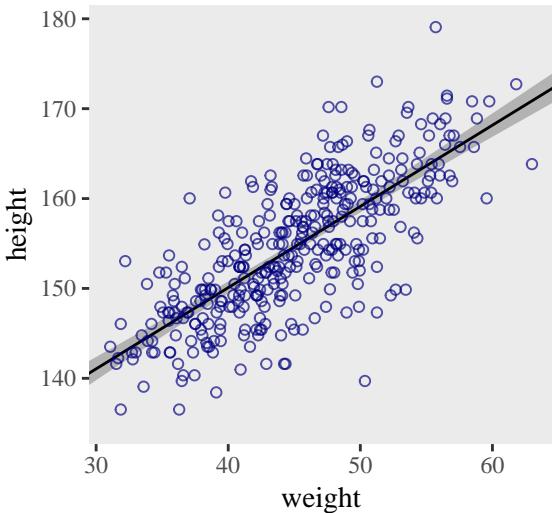
head(mu_summary)
```

```
## # A tibble: 6 x 5
##   Estimate Est.Error Q2.5 Q97.5 weight
##     <dbl>     <dbl> <dbl> <dbl>  <dbl>
## 1     137.      0.864 135. 138.     25
## 2     137.      0.825 136. 139.     26
## 3     138.      0.786 137. 140.     27
```

```
## 4     139.      0.747  138.   141.      28
## 5     140.      0.709  139.   142.      29
## 6     141.      0.671  140.   142.      30
```

Here it is, our analogue to Figure 4.7.b.

```
d2 %>%
  ggplot(aes(x = weight, y = height)) +
  geom_smooth(data = mu_summary,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "grey70", color = "black", alpha = 1, size = 1/2) +
  geom_point(color = "navyblue", shape = 1, size = 1.5, alpha = 2/3) +
  coord_cartesian(xlim = range(d2$weight)) +
  theme(text = element_text(family = "Times"),
        panel.grid = element_blank())
```



And if you wanted to use intervals other than the default 95% ones, you'd enter a `probs` argument like this: `fitted(b4.3, newdata = weight.seq, probs = c(.25, .75))`. The resulting third and fourth vectors from the `fitted()` object would be named `Q25` and `Q75` instead of the default `Q2.5` and `Q97.5`. The `Q` prefix stands for quantile.

4.4.3.4.1 Overthinking: How `link fitted()` works.

Similar to `rethinking::link()`, `brms::fitted()` uses the formula from your model to compute the model expectations for a given set of predictor values. I use it a lot in this project. If you follow along, you'll get a good handle on it.

4.4.3.5 Prediction intervals.

Even though our full statistical model (omitting priors for the sake of simplicity) is

$$h_i \sim \text{Normal}(\mu_i = \alpha + \beta x, \sigma)$$

we've only been plotting the μ part. In order to bring in the variability expressed by σ , we'll have to switch to `predict()`. Much as `brms::fitted()` was our analogue to `rethinking::link()`, `brms::predict()` is our analogue to `rethinking::sim()`.

We can reuse our `weight_seq` data from before. But in case you forgot, here's that code again.

```
weight_seq <- tibble(weight = seq(from = 25, to = 70, by = 1))
```

The `predict()` code looks a lot like what we used for `fitted()`.

```

pred_height <-
  predict(b4.3,
    newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)

pred_height %>%
  slice(1:6)

```

```

## # A tibble: 6 x 5
##   Estimate Est.Error Q2.5 Q97.5 weight
##   <dbl>     <dbl> <dbl> <dbl>   <dbl>
## 1 136.      5.24  126.  147.     25
## 2 137.      5.21  127.  148.     26
## 3 138.      5.25  128.  149.     27
## 4 139.      5.13  129.  149.     28
## 5 140.      5.15  130.  150.     29
## 6 141.      5.07  131.  151.     30

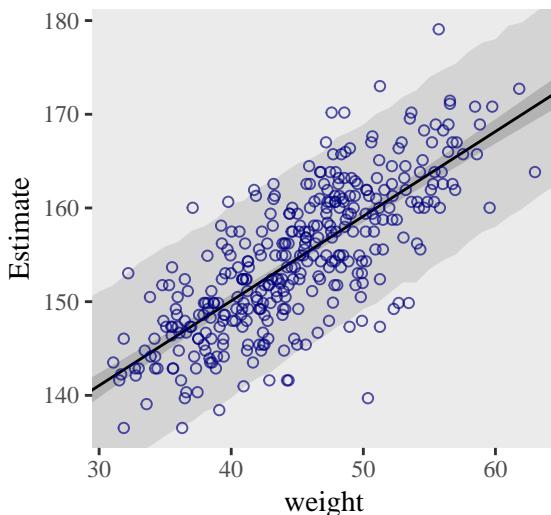
```

This time the summary information in our data frame is for, as McElreath put it, “simulated heights, not distributions of plausible average height, μ ” (p. 108). Another way of saying that is that these simulations are the joint consequence of both μ and σ , unlike the results of `fitted()`, which only reflect μ . Our plot for Figure 4.8:

```

d2 %>%
  ggplot(aes(x = weight)) +
  geom_ribbon(data = pred_height,
    aes(ymin = Q2.5, ymax = Q97.5),
    fill = "grey83") +
  geom_smooth(data = mu_summary,
    aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
    stat = "identity",
    fill = "grey70", color = "black", alpha = 1, size = 1/2) +
  geom_point(aes(y = height),
    color = "navyblue", shape = 1, size = 1.5, alpha = 2/3) +
  coord_cartesian(xlim = range(d2$weight),
    ylim = range(d2$height)) +
  theme(text = element_text(family = "Times"),
    panel.grid = element_blank())

```



4.5 Polynomial regression

Remember d?

```
d %>%
  glimpse()
```

```
## Observations: 544
## Variables: 4
## $ height <dbl> 151.7650, 139.7000, 136.5250, 156.8450, 145.4150, 163.8300, 149.2250, 168.9100, ...
## $ weight <dbl> 47.82561, 36.48581, 31.86484, 53.04191, 41.27687, 62.99259, 38.24348, 55.47997, ...
## $ age     <dbl> 63.0, 63.0, 65.0, 41.0, 51.0, 35.0, 32.0, 27.0, 19.0, 54.0, 47.0, 66.0, 73.0, 20...
## $ male    <int> 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0,...
```

The quadratic is probably the most commonly used polynomial regression model.

$$\mu = \alpha + \beta_1 x_i + \beta_2 x_i^2$$

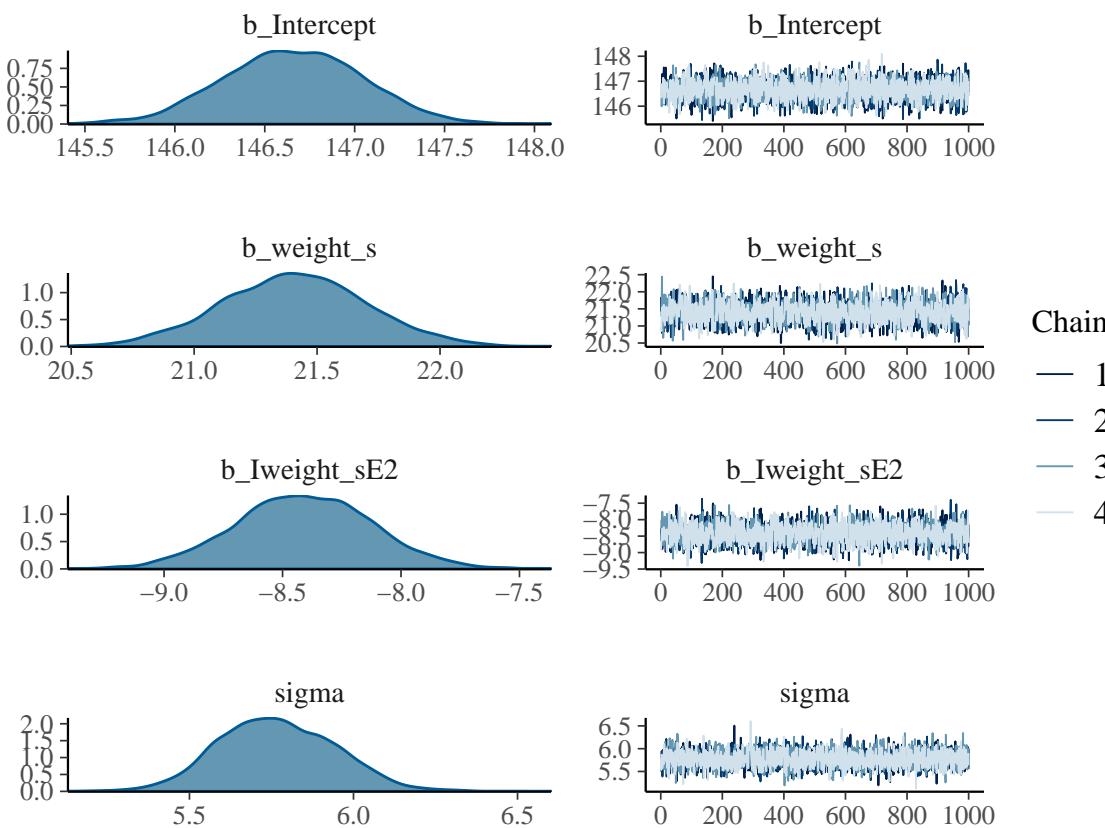
McElreath warned: “Fitting these models to data is easy. Interpreting them can be hard” (p. 111). Standardizing will help `brm()` fit the model. We might standardize our `weight` variable like so.

```
d <-
d %>%
  mutate(weight_s = (weight - mean(weight)) / sd(weight))
```

Here's the quadratic model in brms.

```
b4.5 <-
  brm(data = d, family = gaussian,
       height ~ 1 + weight_s + I(weight_s^2),
       prior = c(prior(normal(178, 100), class = Intercept),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 1), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       seed = 4)
```

```
plot(b4.5)
```



```
print(b4.5)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 1 + weight_s + I(weight_s^2)
##   Data: d (Number of observations: 544)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     146.66      0.39  145.90   147.40      3924  1.00
## weight_s       21.41      0.29   20.83    21.99      2886  1.00
## Iweight_sE2    -8.41      0.29   -8.98    -7.85      3606  1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma        5.77      0.18    5.44     6.11      4089  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Our quadratic plot requires new `fitted()`- and `predict()`-oriented wrangling.

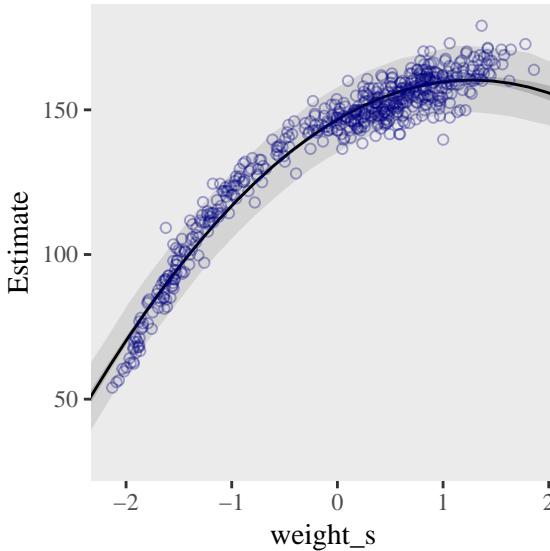
```
weight_seq <- tibble(weight_s = seq(from = -2.5, to = 2.5, length.out = 30))

f_quad <-
  fitted(b4.5,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)
```

```
p_quad <-
  predict(b4.5,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)
```

Behold the code for our version of Figure 4.9.a. You'll notice how little the code changed from that for Figure 4.8, above.

```
ggplot(data = d,
       aes(x = weight_s)) +
  geom_ribbon(data = p_quad,
              aes(ymin = Q2.5, ymax = Q97.5),
              fill = "grey83") +
  geom_smooth(data = f_quad,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "grey70", color = "black", alpha = 1, size = 1/2) +
  geom_point(aes(y = height),
             color = "navyblue", shape = 1, size = 1.5, alpha = 1/3) +
  coord_cartesian(xlim = range(d$weight_s)) +
  theme(text = element_text(family = "Times"),
        panel.grid = element_blank())
```



From a formula perspective, the cubic model is a simple extension of the quadratic:

$$\mu = \alpha + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3$$

Fit it like so.

```
b4.6 <-
  brm(data = d, family = gaussian,
       height ~ 1 + weight_s + I(weight_s^2) + I(weight_s^3),
       prior = c(prior(normal(178, 100), class = Intercept),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 1), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       seed = 4)
```

And now we'll fit the good old linear model.

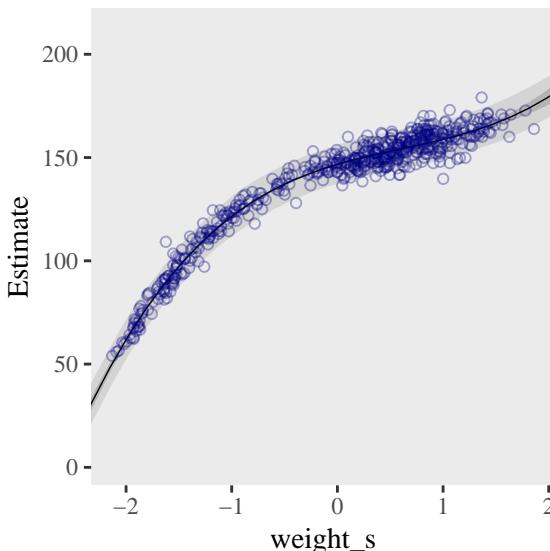
```
b4.7 <-
  brm(data = d, family = gaussian,
    height ~ 1 + weight_s,
    prior = c(prior(normal(178, 100), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 1), class = sigma)),
    iter = 2000, warmup = 1000, chains = 4, cores = 4,
    seed = 4)
```

Here's the `fitted()`, `predict()`, and `ggplot2` code for Figure 4.9.c, the cubic model.

```
f_cub <-
  fitted(b4.6,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)

p_cub <-
  predict(b4.6,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)

ggplot(data = d,
       aes(x = weight_s)) +
  geom_ribbon(data = p_cub,
              aes(ymin = Q2.5, ymax = Q97.5),
              fill = "grey83") +
  geom_smooth(data = f_cub,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "grey70", color = "black", alpha = 1, size = 1/4) +
  geom_point(aes(y = height),
             color = "navyblue", shape = 1, size = 1.5, alpha = 1/3) +
  coord_cartesian(xlim = range(d$weight_s)) +
  theme(text = element_text(family = "Times"),
        panel.grid = element_blank())
```

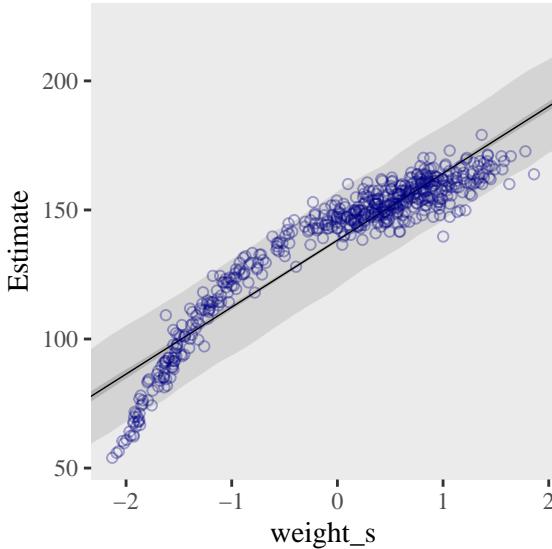


And here's the `fitted()`, `predict()`, and `ggplot2` code for Figure 4.9.a, the linear model.

```
f_line <-
  fitted(b4.7,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)

p_line <-
  predict(b4.7,
         newdata = weight_seq) %>%
  as_tibble() %>%
  bind_cols(weight_seq)

ggplot(data = d,
       aes(x = weight_s)) +
  geom_ribbon(data = p_line,
              aes(ymin = Q2.5, ymax = Q97.5,
                  fill = "grey83")) +
  geom_smooth(data = f_line,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "grey70", color = "black", alpha = 1, size = 1/4) +
  geom_point(aes(y = height),
              color = "navyblue", shape = 1, size = 1.5, alpha = 1/3) +
  coord_cartesian(xlim = range(d$weight_s)) +
  theme(text = element_text(family = "Times"),
        panel.grid = element_blank())
```



4.5.0.0.1 Overthinking: Converting back to natural scale.

You can apply McElreath's conversion trick within the ggplot2 environment, too. Here it is with the cubic model.

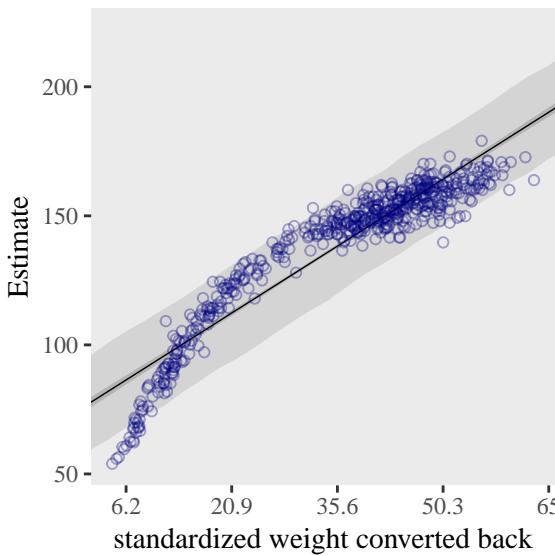
```
at <- c(-2, -1, 0, 1, 2)

ggplot(data = d,
       aes(x = weight_s)) +
  geom_ribbon(data = p_line,
              aes(ymin = Q2.5, ymax = Q97.5,
                  fill = "grey83")) +
  geom_smooth(data = f_line,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
```

```

    fill = "grey70", color = "black", alpha = 1, size = 1/4) +
geom_point(aes(y = height),
            color = "navyblue", shape = 1, size = 1.5, alpha = 1/3) +
coord_cartesian(xlim = range(d$weight_s)) +
theme(text = element_text(family = "Times"),
      panel.grid = element_blank()) +
# here it is!
scale_x_continuous("standardized weight converted back",
                    breaks = at,
                    labels = round(at * sd(d$weight) + mean(d$weight), 1))

```



Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()
```

```

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] grid       parallel   stats      graphics   grDevices  utils      datasets   methods    base
##
## other attached packages:
## [1] tidybayes_1.0.4      brms_2.8.8           Rcpp_1.0.1          rstan_2.18.2

```

```
## [5] StanHeaders_2.18.0-1 forcats_0.3.0      stringr_1.4.0      dplyr_0.8.0.1
## [9] purrr_0.2.5          readr_1.1.1       tidyverse_1.2.1     tibble_2.1.1
## [13] ggplot2_3.1.1        tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2      ggridges_0.5.0      rsconnect_0.8.8
## [4] rprojroot_1.3-2       ggstance_0.3       markdown_0.8
## [7] base64enc_0.1-3       rstudioapi_0.7      svUnit_0.7-12
## [10] DT_0.4                fansi_0.4.0         mvtnorm_1.0-10
## [13] lubridate_1.7.4       xml2_1.2.0         bridgesampling_0.6-0
## [16] knitr_1.20             shinythemes_1.1.1   bayesplot_1.6.0
## [19] jsonlite_1.5           LaplacesDemon_16.1.1 broom_0.5.1
## [22] shiny_1.1.0            compiler_3.5.1      httr_1.3.1
## [25] backports_1.1.4       assertthat_0.2.0    Matrix_1.2-14
## [28] lazyeval_0.2.2         cli_1.0.1          later_0.7.3
## [31] htmltools_0.3.6        prettyunits_1.0.2   tools_3.5.1
## [34] igraph_1.2.1           coda_0.19-2         gtable_0.3.0
## [37] glue_1.3.1.9000        reshape2_1.4.3      cellranger_1.1.0
## [40] nlme_3.1-137          crosstalk_1.0.0    xfun_0.3
## [43] ps_1.2.1               rvest_0.3.2         mime_0.5
## [46] miniUI_0.1.1.1        gtools_3.8.1        MASS_7.3-50
## [49] zoo_1.8-2              scales_1.0.0        colourpicker_1.0
## [52] hms_0.4.2              promises_1.0.1     Brobdingnag_1.2-6
## [55] inline_0.3.15          shinystan_2.5.0    yaml_2.1.19
## [58] gridExtra_2.3           loo_2.1.0          stringi_1.4.3
## [61] dygraphs_1.1.1.5        pkgbuild_1.0.2     rlang_0.3.4
## [64] pkgconfig_2.0.2         matrixStats_0.54.0 HDInterval_0.2.0
## [67] evaluate_0.10.1         lattice_0.20-35   rstantools_1.5.1
## [70] htmlwidgets_1.2         labeling_0.3        processx_3.2.1
## [73] tidyselect_0.2.5        plyr_1.8.4          magrittr_1.5
## [76] bookdown_0.9            R6_2.3.0            generics_0.0.2
## [79] pillar_1.3.1            haven_1.1.2        withr_2.1.2
## [82] xts_0.10-2              abind_1.4-5         modelr_0.1.2
## [85] crayon_1.3.4            arrayhelpers_1.0-20160527 utf8_1.1.4
## [88] rmarkdown_1.10           readxl_1.1.0        callr_3.1.0
## [91] threejs_0.3.1           digest_0.6.18       xtable_1.8-2
## [94] httpuv_1.4.4.2          stats4_3.5.1        munsell_0.5.0
## [97] viridisLite_0.3.0        shinyjs_1.0
```


Chapter 5

Multivariate Linear Models

McElreath's listed reasons for multivariable regression include:

- statistical control for confounds
- multiple causation
- interactions

We'll approach the first two in this chapter. Interactions are reserved for Chapter 6.

5.1 Spurious associations

Load the Waffle House data.

```
library(rethinking)
data(WaffleDivorce)
d <- WaffleDivorce
```

Unload rethinking and load brms and, while we're at it, the tidyverse.

```
rm(WaffleDivorce)
detach(package:rethinking, unload = T)
library(brms)
library(tidyverse)
```

I'm not going to show the output, but you might go ahead and investigate the data with the typical functions. E.g.,

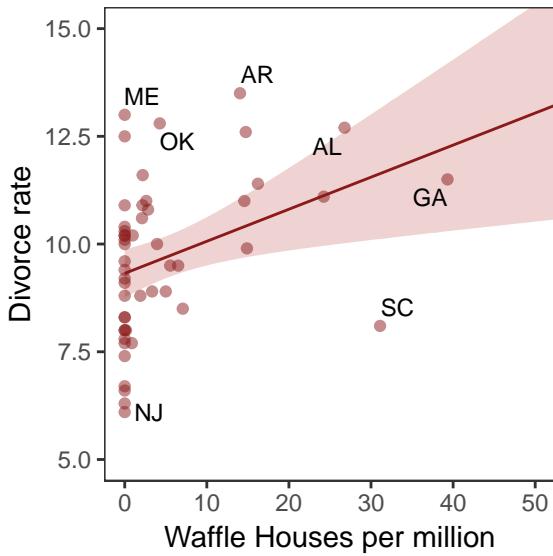
```
head(d)
glimpse(d)
```

Now we have our data, we can reproduce Figure 5.1. One convenient way to get the handful of state labels into the plot was with the `geom_text_repel()` function from the `ggrepel` package. But first, we spent the last few chapters warming up with ggplot2. Going forward, each chapter will have its own plot theme. In this chapter, we'll characterize the plots with `theme_bw() + theme(panel.grid = element_rect())` and coloring based off of "firebrick".

```
# install.packages("ggrepel", dependencies = T)
library(ggrepel)

d %>%
  ggplot(aes(x = WaffleHouses/Population, y = Divorce)) +
  stat_smooth(method = "lm", fullrange = T, size = 1/2,
             color = "firebrick4", fill = "firebrick", alpha = 1/5) +
```

```
geom_point(size = 1.5, color = "firebrick4", alpha = 1/2) +
  geom_text_repel(data = d %>% filter(Loc %in% c("ME", "OK", "AR", "AL", "GA", "SC", "NJ")),
    aes(label = Loc),
    size = 3, seed = 1042) + # this makes it reproducible
  scale_x_continuous("Waffle Houses per million", limits = c(0, 55)) +
  coord_cartesian(xlim = 0:50, ylim = 5:15) +
  ylab("Divorce rate") +
  theme_bw() +
  theme(panel.grid = element_blank())
```

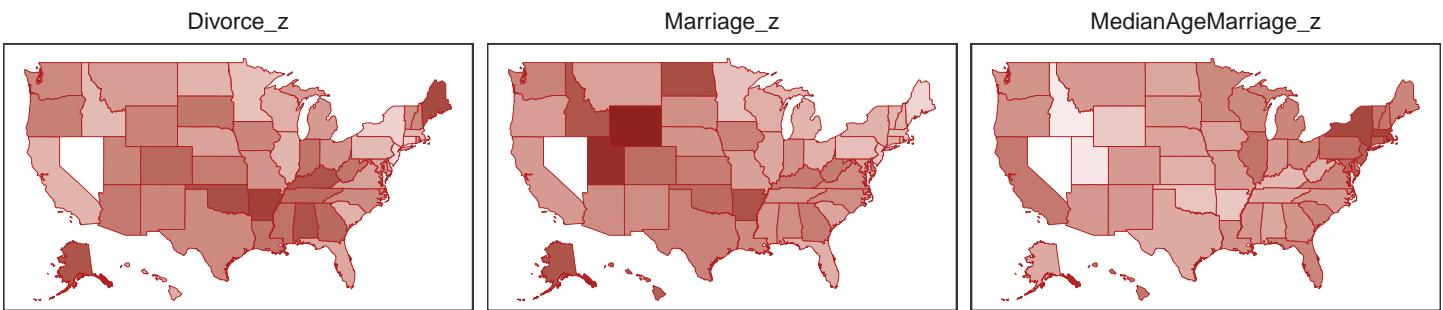


With `coord_map()` and help from the [fiftystater package](#) (which gives us access to lat/long data for all fifty states via `fifty_states`), we can plot our three major variables in a map format.

```
library(fiftystater)

d %>%
  # first we'll standardize the three variables to put them all on the same scale
  mutate(Divorce_z           = (Divorce - mean(Divorce)) / sd(Divorce),
         MedianAgeMarriage_z = (MedianAgeMarriage - mean(MedianAgeMarriage)) / sd(MedianAgeMarriage),
         Marriage_z          = (Marriage - mean(Marriage)) / sd(Marriage),
         # need to make the state names lowercase to match with the map data
         Location             = str_to_lower(Location)) %>%
  # here we select the relevant variables and put them in the long format to facet with `facet_wrap()`
  select(Divorce_z:Marriage_z, Location) %>%
  gather(key, value, -Location) %>%

  ggplot(aes(map_id = Location)) +
  geom_map(aes(fill = value), map = fifty_states,
    color = "firebrick", size = 1/15) +
  expand_limits(x = fifty_states$long, y = fifty_states$lat) +
  scale_x_continuous(NULL, breaks = NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  scale_fill_gradient(low = "#f8eaea", high = "firebrick4") +
  coord_map() +
  theme_bw() +
  theme(panel.grid      = element_blank(),
        legend.position = "none",
        strip.background = element_rect(fill = "transparent", color = "transparent")) +
  facet_wrap(~key)
```



One of the advantages of this visualization method is it just became clear that Nevada is missing from the `WaffleDivorce` data. Execute `d %>% distinct(Location)` to see for yourself. Those missing data should motivate the skills we'll cover in Chapter 14. But let's get back on track.

Here we'll officially standardize the predictor, `MedianAgeMarriage`.

```
d <-
d %>%
  mutate(MedianAgeMarriage_s = (MedianAgeMarriage - mean(MedianAgeMarriage)) /
    sd(MedianAgeMarriage))
```

Now we're ready to fit the first univariable model.

```
b5.1 <-
  brm(data = d, family = gaussian,
       Divorce ~ 1 + MedianAgeMarriage_s,
       prior = c(prior(normal(10, 10), class = Intercept),
                 prior(normal(0, 1), class = b),
                 prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

Check the summary.

```
print(b5.1)

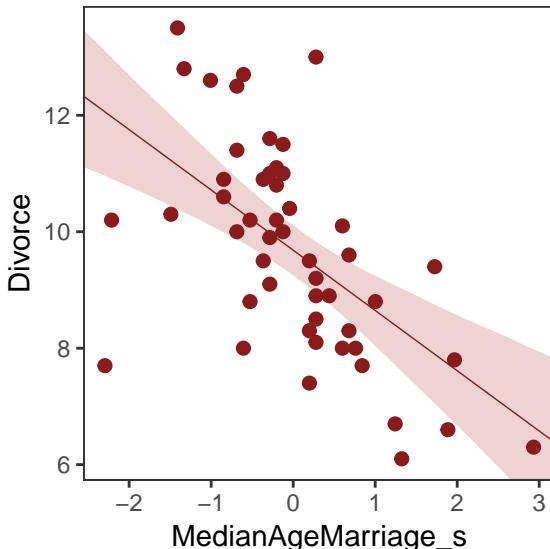
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: Divorce ~ 1 + MedianAgeMarriage_s
##   Data: d (Number of observations: 50)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##           total post-warmup samples = 6000
##
## Population-Level Effects:
##                               Estimate Est.Error 1-95% CI  u-95% CI Eff.Sample Rhat
## Intercept                  9.68     0.22     9.25    10.11      5675 1.00
## MedianAgeMarriage_s      -1.04     0.22    -1.45    -0.59      5945 1.00
##
## Family Specific Parameters:
##                               Estimate Est.Error 1-95% CI  u-95% CI Eff.Sample Rhat
## sigma                   1.52     0.16     1.24     1.87      4029 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

We'll employ `fitted()` to make Figure 5.2.b. In preparation for `fitted()` we'll make a new tibble, `nd`, composed of a handful of densely-packed values for our predictor, `MedianAgeMarriage_s`. With the `newdata` argument, we'll use those values to return model-implied expected values for `Divorce`.

```
# define the range of `MedianAgeMarriage_s` values we'd like to feed into `fitted()`
nd <- tibble(MedianAgeMarriage_s = seq(from = -3, to = 3.5, length.out = 30))

# now use `fitted()` to get the model-implied trajectories
f <-
  fitted(b5.1, newdata = nd) %>%
  as_tibble() %>%
  # tack the `nd` data onto the `fitted()` results
  bind_cols(nd)

# plot
ggplot(data = f,
        aes(x = MedianAgeMarriage_s, y = Estimate)) +
  geom_smooth(aes(ymax = Q2.5, ymin = Q97.5),
              stat = "identity",
              fill = "firebrick", color = "firebrick4", alpha = 1/5, size = 1/4) +
  geom_point(data = d,
             aes(y = Divorce),
             size = 2, color = "firebrick4") +
  ylab("Divorce") +
  coord_cartesian(xlim = range(d$MedianAgeMarriage_s),
                  ylim = range(d$Divorce)) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



Before fitting the next model, we'll standardize Marriage.

```
d <-
d %>%
  mutate(Marriage_s = (Marriage - mean(Marriage)) / sd(Marriage))
```

We're ready to fit our second univariable model.

```
b5.2 <-
  brm(data = d, family = gaussian,
       Divorce ~ 1 + Marriage_s,
       prior = c(prior(normal(10, 10), class = Intercept),
                 prior(normal(0, 1), class = b),
                 prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

```
print(b5.2)
```

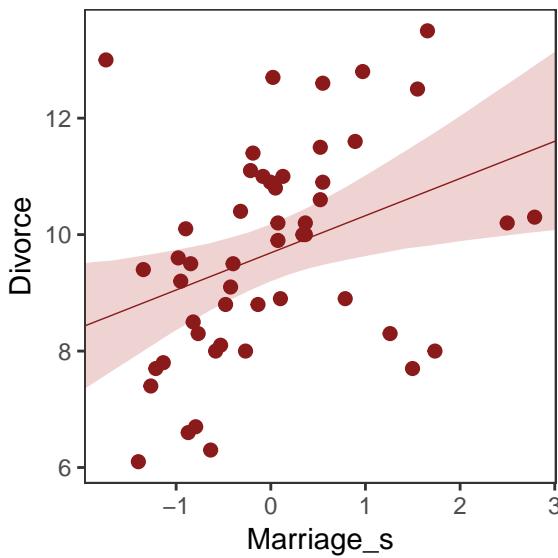
```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: Divorce ~ 1 + Marriage_s
##   Data: d (Number of observations: 50)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept      9.69     0.25    9.20    10.18      5516 1.00
## Marriage_s      0.64     0.25    0.16    1.12      5229 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.75     0.18    1.44    2.15      4519 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Now we'll wangle and plot our version of Figure 5.2.a.

```
nd <- tibble(Marriage_s = seq(from = -2.5, to = 3.5, length.out = 30))

f <-
  fitted(b5.2, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

ggplot(data = f,
       aes(x = Marriage_s, y = Estimate)) +
  geom_smooth(aes(ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "firebrick", color = "firebrick4", alpha = 1/5, size = 1/4) +
  geom_point(data = d,
             aes(y = Divorce),
             size = 2, color = "firebrick4") +
  coord_cartesian(xlim = range(d$Marriage_s),
                  ylim = range(d$Divorce)) +
  ylab("Divorce") +
  theme_bw() +
  theme(panel.grid = element_blank())
```



But merely comparing parameter means between different bivariate regressions is no way to decide which predictor is better. Both of these predictors could provide independent value, or they could be redundant, or one could eliminate the value of the other. So we'll build a multivariate model with the goal of measuring the partial value of each predictor. The question we want answered is:

What is the predictive value of a variable, once I already know all of the other predictor variables?
(p. 123, emphasis in the original)

5.1.1 Multivariate notation.

Now we'll get both predictors in there with our very first multivariable model. We can write the statistical model as

$$\begin{aligned} \text{Divorce}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta_1 \text{Marriage_s}_i + \beta_2 \text{MedianAgeMarriage_s}_i \\ \alpha &\sim \text{Normal}(10, 10) \\ \beta_1 &\sim \text{Normal}(0, 1) \\ \beta_2 &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{Uniform}(0, 10) \end{aligned}$$

It might help to read the + symbols as “or” and then say: *A State’s divorce rate can be a function of its marriage rate or its median age at marriage.* The “or” indicates independent associations, which may be purely statistical or rather causal. (p. 124, emphasis in the original)

5.1.2 Fitting the model.

Much like we used the + operator to add single predictors to the intercept, we just use more + operators in the formula argument to add more predictors. Also notice we're using the same prior `prior(normal(0, 1), class = b)` for both predictors. Within the brms framework, they are both of `class = b`. But if we wanted their priors to differ, we'd make two `prior()` statements and differentiate them with the `coef` argument. You'll see examples of that later on.

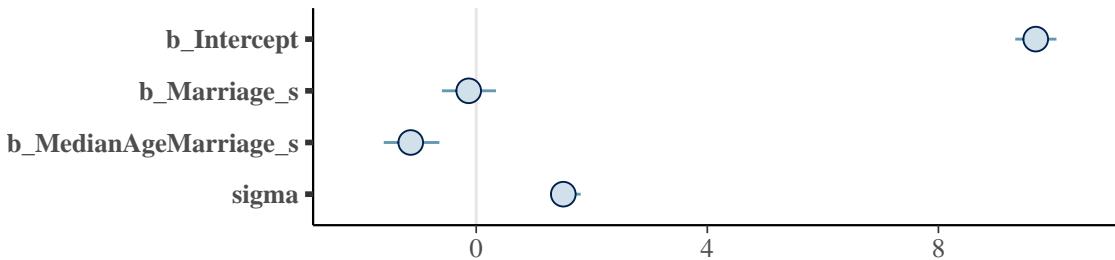
```
b5.3 <-  
  brm(data = d, family = gaussian,  
        Divorce ~ 1 + Marriage_s + MedianAgeMarriage_s,  
        prior = c(prior(normal(10, 10), class = Intercept),  
                  prior(normal(0, 1), class = b),  
                  prior(uniform(0, 10), class = sigma)),  
        iter = 2000, warmup = 500, chains = 4, cores = 4,  
        seed = 5)
```

```
print(b5.3)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: Divorce ~ 1 + Marriage_s + MedianAgeMarriage_s
## Data: d (Number of observations: 50)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept      9.68     0.22    9.26   10.11      5396 1.00
## Marriage_s     -0.13     0.29   -0.68    0.43      3810 1.00
## MedianAgeMarriage_s -1.13     0.29   -1.70   -0.54      3908 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.52     0.16    1.24    1.88      4923 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

The `stanplot()` function is an easy way to get a default coefficient plot. You just put the `brmsfit` object into the function.

```
stanplot(b5.3)
```



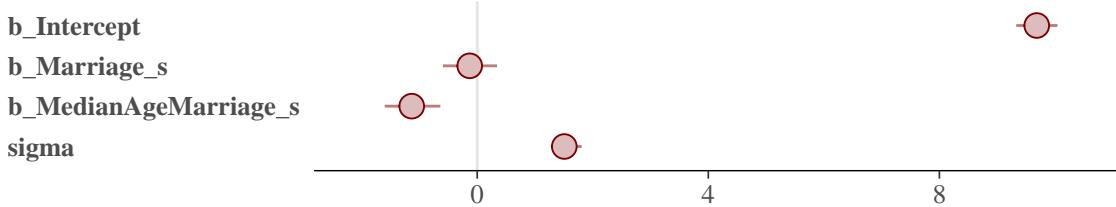
There are numerous ways to make a coefficient plot. Another is with the `mcmc_intervals()` function from the [bayesplot package](#). A nice feature of the `bayesplot` package is its convenient way to alter the color scheme with the `color_scheme_set()` function. Here, for example, we'll make the theme `red`. But note how the `mcmc_intervals()` function requires you to work with the `posterior_samples()` instead of the `brmsfit` object.

```
# install.packages("bayesplot", dependencies = T)
library(bayesplot)

post <- posterior_samples(b5.3)

color_scheme_set("red")
mcmc_intervals(post[, 1:4],
               prob = .5,
               point_est = "median") +
  labs(title = "My fancy bayesplot-based coefficient plot") +
  theme(axis.text.y = element_text(hjust = 0),
        axis.line.x = element_line(size = 1/4),
        axis.line.y = element_blank(),
        axis.ticks.y = element_blank())
```

My fancy bayesplot-based coefficient plot



Because `bayesplot` produces a `ggplot2` object, the plot was adjustable with familiar `ggplot2` syntax. For more ideas, check out [this vignette](#).

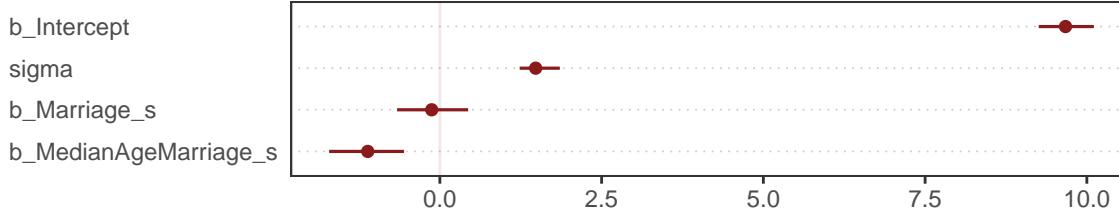
The `tidybaes::stat_pointintervalh()` function offers a third way, this time with a more ground-up `ggplot2` workflow.

```
library(tidybayes)

post %>%
  select(-lp_) %>%
  gather() %>%

  ggplot(aes(x = value, y = reorder(key, value))) + # note how we used `reorder()` to arrange the coefficient
  geom_vline(xintercept = 0, color = "firebrick4", alpha = 1/10) +
  stat_pointintervalh(point_interval = mode_hdi, .width = .95,
                       size = 3/4, color = "firebrick4") +
  labs(title = "My tidybayes-based coefficient plot",
       x = NULL, y = NULL) +
  theme_bw() +
  theme(panel.grid    = element_blank(),
        panel.grid.major.y = element_line(color = alpha("firebrick4", 1/4), linetype = 3),
        axis.text.y   = element_text(hjust = 0),
        axis.ticks.y  = element_blank())
```

My tidybayes-based coefficient plot



The substantive interpretation of all those coefficient plots is: “Once we know median age at marriage for a State, there is little or no additive predictive power in also knowing the rate of marriage in that State” (p. 126, *emphasis* in the original).

5.1.3 Plotting multivariate posteriors.

McElreath’s prose is delightfully deflationary. “There is a huge literature detailing a variety of plotting techniques that all attempt to help one understand multiple linear regression. None of these techniques is suitable for all jobs, and most do not generalize beyond linear regression” (p. 126). Now you’re inspired, let’s learn three:

- Predictor residual plots
- Counterfactual plots
- Posterior prediction plots

5.1.3.1 Predictor residual plots.

To get ready to make our residual plots, we’ll predict `Marriage_s` with `MedianAgeMarriage_s`.

```
b5.4 <-
  brm(data = d, family = gaussian,
    Marriage_s ~ 1 + MedianAgeMarriage_s,
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(uniform(0, 10), class = sigma)),
    iter = 2000, warmup = 500, chains = 4, cores = 4,
    seed = 5)

print(b5.4)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: Marriage_s ~ 1 + MedianAgeMarriage_s
##   Data: d (Number of observations: 50)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept      0.00     0.10   -0.20    0.20      5540 1.00
## MedianAgeMarriage_s  -0.71     0.10   -0.91   -0.51      4941 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       0.72     0.08   0.59    0.89      5352 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

With `fitted()`, we compute the expected values for each state (with the exception of Nevada). Since the `MedianAgeMarriage_s` values for each state are in the date we used to fit the model, we'll omit the `newdata` argument.

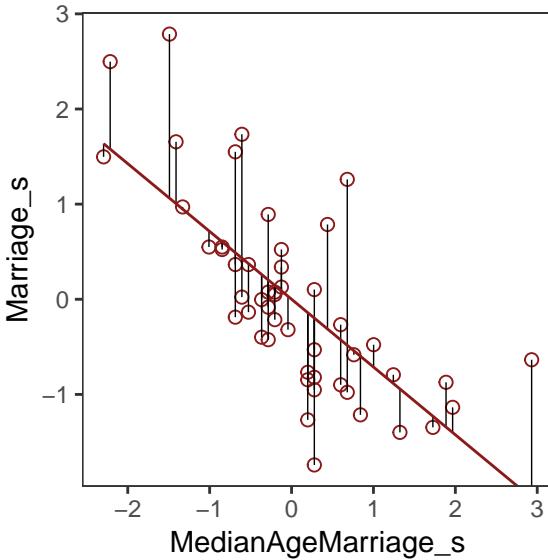
```
f <-
  fitted(b5.4) %>%
  as_tibble() %>%
  bind_cols(d)

head(f)

## # A tibble: 6 x 19
##   Estimate Est.Error   Q2.5   Q97.5 Location Loc  Population
##   <dbl>     <dbl>  <dbl>   <dbl> <fct>   <fct>     <dbl>
## 1  0.433     0.119  2.00e-1  0.667 Alabama  AL      4.78
## 2  0.490     0.124  2.48e-1  0.733 Alaska  AK      0.71
## 3  0.146     0.105 -5.52e-2  0.354 Arizona AZ      6.33
## 4  1.01      0.176  6.56e-1  1.35   Arkansas AR      2.92
## 5  -0.427    0.121 -6.65e-1 -0.194 Califor~ CA      37.2
## 6  0.203     0.107 -8.95e-4  0.414 Colorado CO      5.03
## # ... with 12 more variables: MedianAgeMarriage <dbl>, Marriage <dbl>,
## #   Marriage.SE <dbl>, Divorce <dbl>, Divorce.SE <dbl>,
## #   WaffleHouses <int>, South <int>, Slaves1860 <int>,
## #   Population1860 <int>, PropSlaves1860 <dbl>, MedianAgeMarriage_s <dbl>,
## #   Marriage_s <dbl>
```

After a little data processing, we can make Figure 5.3.

```
f %>%
  ggplot(aes(x = MedianAgeMarriage_s, y = Marriage_s)) +
  geom_point(size = 2, shape = 1, color = "firebrick4") +
  geom_segment(aes(xend = MedianAgeMarriage_s, yend = Estimate),
               size = 1/4) +
  geom_line(aes(y = Estimate),
            color = "firebrick4") +
  coord_cartesian(ylim = range(d$Marriage_s)) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



We get the residuals with the well-named `residuals()` function. Much like with `brms::fitted()`, `brms::residuals()` returns a four-vector matrix with the number of rows equal to the number of observations in the original data (by default, anyway). The vectors have the familiar names: `Estimate`, `Est.Error`, `Q2.5`, and `Q97.5`. See the [brms reference manual](#) for details.

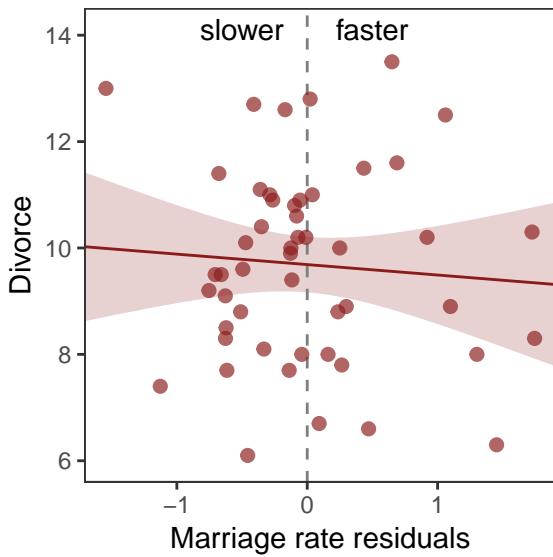
With our residuals in hand, we just need a little more data processing to make Figure 5.4.a.

```
r <- 
  residuals(b5.4) %>%
  # to use this in ggplot2, we need to make it a tibble or data frame
  as_tibble() %>%
  bind_cols(d)

# for the annotation at the top
text <-
  tibble(Estimate = c(-0.5, 0.5),
         Divorce = 14.1,
         label = c("slower", "faster"))

# plot
r %>%
  ggplot(aes(x = Estimate, y = Divorce)) +
  stat_smooth(method = "lm", fullrange = T,
              color = "firebrick4", fill = "firebrick4",
              alpha = 1/5, size = 1/2) +
  geom_vline(xintercept = 0, linetype = 2, color = "grey50") +
  geom_point(size = 2, color = "firebrick4", alpha = 2/3) +
  geom_text(data = text,
            aes(label = label)) +
  scale_x_continuous("Marriage rate residuals", limits = c(-2, 2)) +
```

```
coord_cartesian(xlim = range(r$Estimate),
                 ylim = c(6, 14.1)) +
theme_bw() +
theme(panel.grid = element_blank())
```



To get the MedianAgeMarriage_s residuals, we have to fit the corresponding model first.

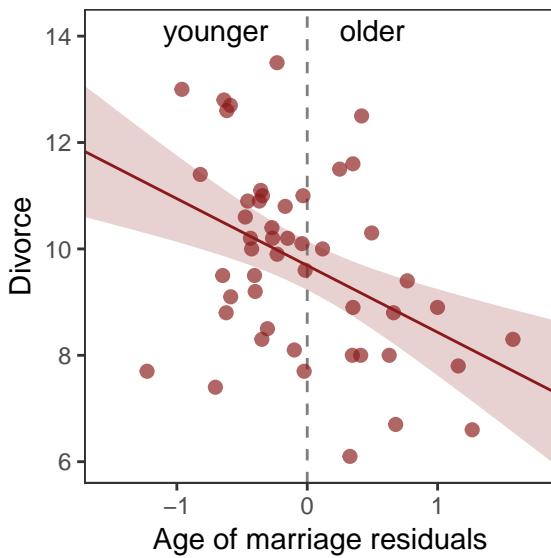
```
b5.4b <-
brm(data = d, family = gaussian,
MedianAgeMarriage_s ~ 1 + Marriage_s,
prior = c(prior(normal(0, 10), class = Intercept),
          prior(normal(0, 1), class = b),
          prior(uniform(0, 10), class = sigma)),
iter = 2000, warmup = 500, chains = 4, cores = 4,
seed = 5)
```

And now we'll get the new batch of residuals, do a little data processing, and make a plot corresponding to Figure 5.4.b.

```
text <-
tibble(Estimate = c(-0.7, 0.5),
      Divorce = 14.1,
      label = c("younger", "older"))

residuals(b5.4b) %>%
as_tibble() %>%
bind_cols(d) %>%

ggplot(aes(x = Estimate, y = Divorce)) +
stat_smooth(method = "lm", fullrange = T,
            color = "firebrick4", fill = "firebrick4",
            alpha = 1/5, size = 1/2) +
geom_vline(xintercept = 0, linetype = 2, color = "grey50") +
geom_point(size = 2, color = "firebrick4", alpha = 2/3) +
geom_text(data = text,
          aes(label = label)) +
scale_x_continuous("Age of marriage residuals", limits = c(-2, 3)) +
coord_cartesian(xlim = range(r$Estimate),
                 ylim = c(6, 14.1)) +
theme_bw() +
theme(panel.grid = element_blank())
```



5.1.3.2 Counterfactual plots.

A second sort of inferential plot displays the implied predictions of the model. I call these plots *counterfactual*, because they can be produced for any values of the predictor variable you like, even unobserved or impossible combinations like very high median age of marriage and very high marriage rate. There are no States with this combination, but in a counterfactual plot, you can ask the model for a prediction for such a State. (p. 129, *emphasis* in the original)

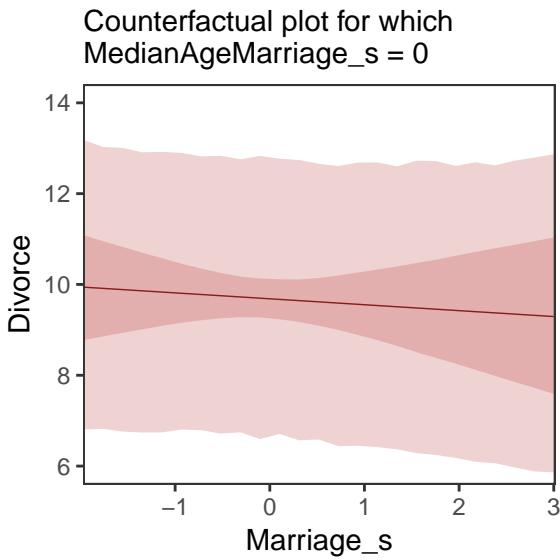
Making Figure 5.5.a requires a little more data wrangling than before.

```
# we need new `nd` data
nd <-
  tibble(Marriage_s      = seq(from = -3, to = 3, length.out = 30),
         MedianAgeMarriage_s = mean(d$MedianAgeMarriage_s))

fitted(b5.3, newdata = nd) %>%
  as_tibble() %>%
  # since `fitted()` and `predict()` name their intervals the same way,
  # we'll need to `rename()` them to keep them straight
  rename(f_ll = Q2.5,
         f_ul = Q97.5) %>%
  # note how we're just nesting the `predict()` code right inside `bind_cols()`
  bind_cols(
    predict(b5.3, newdata = nd) %>%
      as_tibble() %>%
      # since we only need the intervals, we'll use `transmute()` rather than `mutate()`
      transmute(p_ll = Q2.5,
                p_ul = Q97.5),
    # now tack on the `nd` data
    nd) %>%

# we're finally ready to plot
ggplot(aes(x = Marriage_s, y = Estimate)) +
  geom_ribbon(aes(ymin = p_ll, ymax = p_ul),
              fill = "firebrick", alpha = 1/5) +
  geom_smooth(aes(ymin = f_ll, ymax = f_ul),
              stat = "identity",
              fill = "firebrick", color = "firebrick4", alpha = 1/5, size = 1/4) +
  coord_cartesian(xlim = range(d$Marriage_s),
                  ylim = c(6, 14)) +
  labs(subtitle = "Counterfactual plot for which\nMedianAgeMarriage_s = 0",
```

```
y = "Divorce") +
theme_bw() +
theme(panel.grid = element_blank())
```

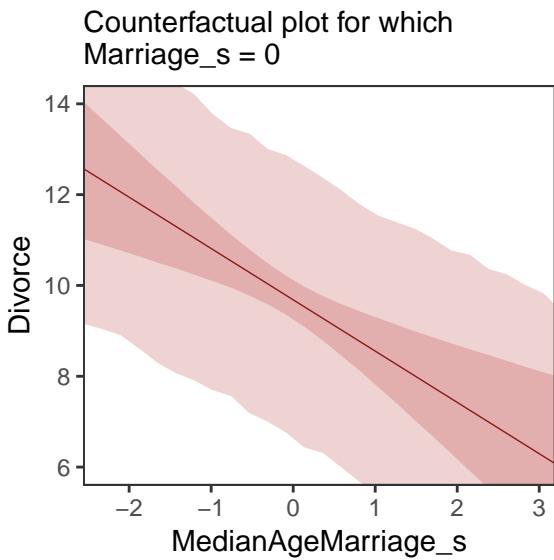


We follow the same process for Figure 5.5.b.

```
# new data
nd <-
  tibble(MedianAgeMarriage_s = seq(from = -3, to = 3.5, length.out = 30),
         Marriage_s           = mean(d$Marriage_s))

# `fitted()` + `predict()`
fitted(b5.3, newdata = nd) %>%
  as_tibble() %>%
  rename(f_ll = Q2.5,
         f_ul = Q97.5) %>%
  bind_cols(
    predict(b5.3, newdata = nd) %>%
      as_tibble() %>%
      transmute(p_ll = Q2.5,
                p_ul = Q97.5),
    nd
  ) %>%

# plot
ggplot(aes(x = MedianAgeMarriage_s, y = Estimate)) +
  geom_ribbon(aes(ymin = p_ll, ymax = p_ul),
              fill = "firebrick", alpha = 1/5) +
  geom_smooth(aes(ymin = f_ll, ymax = f_ul),
              stat = "identity",
              fill = "firebrick", color = "firebrick4", alpha = 1/5, size = 1/4) +
  coord_cartesian(xlim = range(d$MedianAgeMarriage_s),
                  ylim = c(6, 14)) +
  labs(subtitle = "Counterfactual plot for which\nMarriage_s = 0",
       y = "Divorce") +
  theme_bw() +
  theme(panel.grid = element_blank())
```



A tension with such plots, however, lies in their counterfactual nature. In the small world of the model, it is possible to change median age of marriage without also changing the marriage rate. But is this also possible in the large world of reality? Probably not...

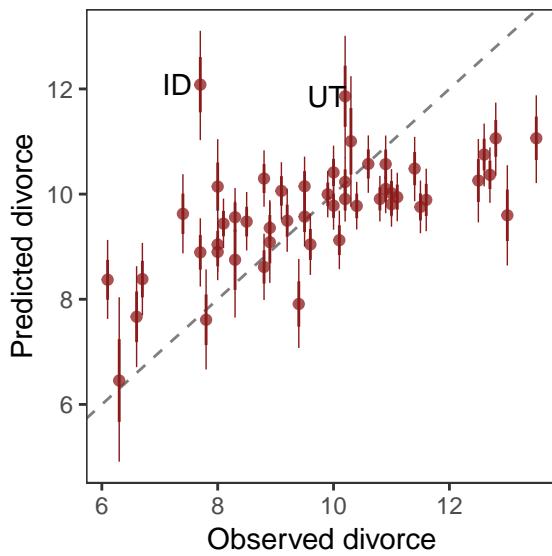
...If our goal is to intervene in the world, there may not be any realistic way to manipulate each predictor without also manipulating the others. This is a serious obstacle to applied science, whether you are an ecologist, an economist, or an epidemiologist [or a psychologist] (p. 131)

5.1.3.3 Posterior prediction plots.

"In addition to understanding the estimates, it's important to check the model fit against the observed data" (p. 131). For more on the topic, check out Gabry and colleagues' *Visualization in Bayesian workflow* or Simpson's related blog post *Touch me, I want to feel your data*.

In this version of Figure 5.6.a, the thin lines are the 95% intervals and the thicker lines are +/- the posterior SD, both of which are returned when you use `fitted()`.

```
fitted(b5.3) %>%
  as_tibble() %>%
  bind_cols(d) %>%
  ggplot(aes(x = Divorce, y = Estimate)) +
  geom_abline(linetype = 2, color = "grey50", size = .5) +
  geom_point(size = 1.5, color = "firebrick4", alpha = 3/4) +
  geom_linerange(aes(ymin = Q2.5, ymax = Q97.5),
                 size = 1/4, color = "firebrick4") +
  geom_linerange(aes(ymin = Estimate - Est.Error,
                     ymax = Estimate + Est.Error),
                 size = 1/2, color = "firebrick4") +
  # Note our use of the dot placeholder, here: https://magrittr.tidyverse.org/reference/pipe.html
  geom_text(data = . %>% filter(Loc %in% c("ID", "UT")),
            aes(label = Loc),
            hjust = 0, nudge_x = - 0.65) +
  labs(x = "Observed divorce",
       y = "Predicted divorce") +
  theme_bw() +
  theme(panel.grid = element_blank())
```



In order to make Figure 5.6.b, we need to clarify the relationships among `fitted()`, `predict()`, and `residuals()`. Here's my attempt in a table.

```
tibble(`brms function` = c("fitted", "predict", "residual"),
      mean = c("same as the data", "same as the data", "in a deviance-score metric"),
      scale = c("excludes sigma", "includes sigma", "excludes sigma")) %>%
knitr::kable()
```

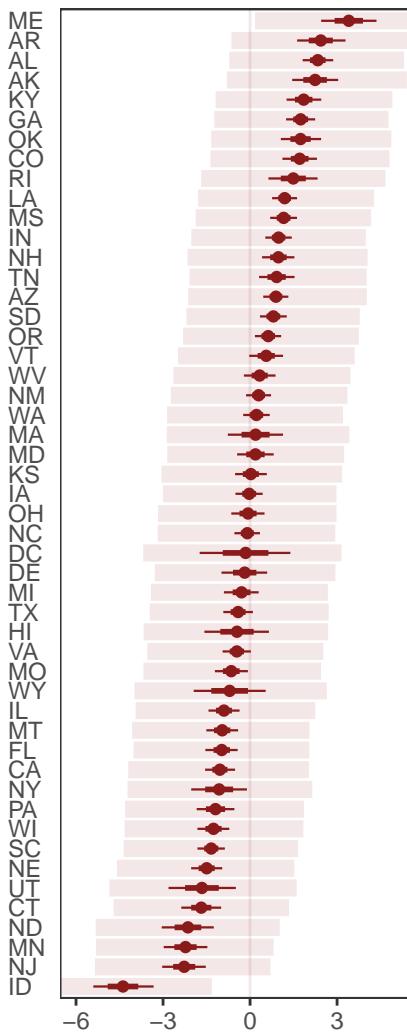
brms function	mean	scale
fitted	same as the data	excludes sigma
predict	same as the data	includes sigma
residual	in a deviance-score metric	excludes sigma

Hopefully this clarifies that if we want to incorporate the prediction interval in a deviance metric, we'll need to first use `predict()` and then subtract the intervals from their corresponding Divorce values in the data.

```
residuals(b5.3) %>%
  as_tibble() %>%
  rename(f_ll = Q2.5,
         f_ul = Q97.5) %>%
  bind_cols(
    predict(b5.3) %>%
      as_tibble() %>%
      transmute(p_ll = Q2.5,
                p_ul = Q97.5),
    d
  ) %>%
# here we put our `predict()` intervals into a deviance metric
  mutate(p_ll = Divorce - p_ll,
         p_ul = Divorce - p_ul) %>%

# now plot!
  ggplot(aes(x = reorder(Loc, Estimate), y = Estimate)) +
  geom_hline(yintercept = 0, size = 1/2,
             color = "firebrick4", alpha = 1/10) +
  geom_pointrange(aes(ymax = f_ll, ymin = f_ul),
                  size = 2/5, shape = 20, color = "firebrick4") +
  geom_segment(aes(y = Estimate - Est.Error,
                  yend = Estimate + Est.Error,
                  x = Loc,
                  xend = Loc),
               size = 1, color = "firebrick4") +
```

```
geom_segment(aes(y      = p_ll,
                  yend = p_ul,
                  x      = Loc,
                  xend = Loc),
               size = 3, color = "firebrick4", alpha = 1/10) +
  labs(x = NULL, y = NULL) +
  coord_flip(ylim = c(-6, 5)) +
  theme_bw() +
  theme(panel.grid    = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y  = element_text(hjust = 0))
```

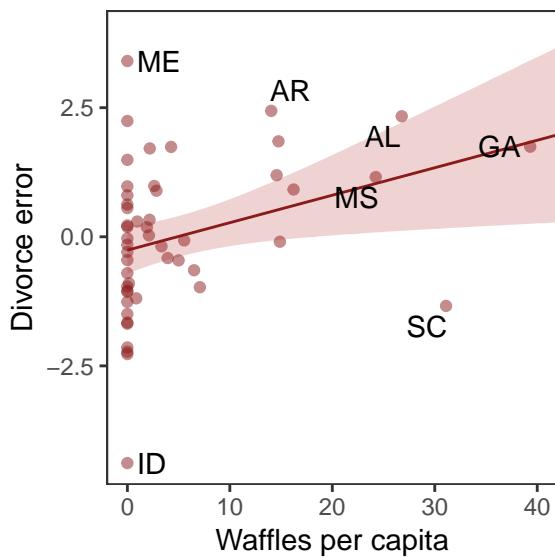


Compared to the last couple plots, Figure 5.6.c is pretty simple.

```
residuals(b5.3) %>%
  as_tibble() %>%
  bind_cols(d) %>%
  mutate(wpc = WaffleHouses / Population) %>%

  ggplot(aes(x = wpc, y = Estimate)) +
  geom_point(size = 1.5, color = "firebrick4", alpha = 1/2) +
  stat_smooth(method = "lm", fullrange = T,
              color = "firebrick4", size = 1/2,
              fill = "firebrick", alpha = 1/5) +
  geom_text_repel(data = . %>% filter(Loc %in% c("ME", "AR", "MS", "AL", "GA", "SC", "ID")),
                 aes(label = Loc),
                 seed = 5.6) +
```

```
scale_x_continuous("Waffles per capita", limits = c(0, 45)) +
coord_cartesian(xlim = range(0, 40)) +
ylab("Divorce error") +
theme_bw() +
theme(panel.grid = element_blank())
```



More McElreath inspiration: “No matter how many predictors you’ve already included in a regression, it’s still possible to find spurious correlations with the remaining variation” (p. 134). To keep our deflation train going, it’s worthwhile to repeat the message in McElreath’s **Rethinking: Stats, huh, yeah what is it good for?** box.

Often people want statistical modeling to do things that statistical modeling cannot do. For example, we’d like to know whether an effect is real or rather spurious. Unfortunately, modeling merely quantifies uncertainty in the precise way that the model understands the problem. Usually answers to large world questions about truth and causation depend upon information not included in the model. For example, any observed correlation between an outcome and predictor could be eliminated or reversed once another predictor is added to the model. But if we cannot think of another predictor, we might never notice this. Therefore all statistical models are vulnerable to and demand critique, regardless of the precision of their estimates and apparent accuracy of their predictions. (p. 134)

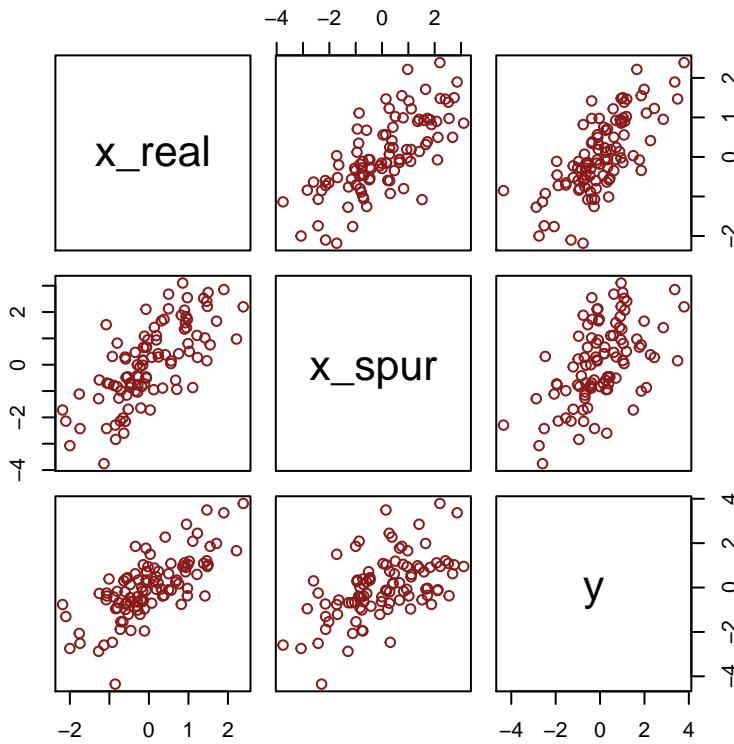
5.1.3.4 Overthinking: Simulating spurious association.

```
n <- 100 # number of cases

set.seed(5) # setting the seed makes the results reproducible
d <-
  tibble(x_real = rnorm(n), # x_real as Gaussian with mean 0 and SD 1 (i.e., the defaults)
         x_spur = rnorm(n, x_real), # x_spur as Gaussian with mean = x_real
         y =       rnorm(n, x_real)) # y as Gaussian with mean = x_real
```

Here are the quick `pairs()` plots.

```
pairs(d, col = "firebrick4")
```



We may as well fit a model.

```
brm(data = d, family = gaussian,
     y ~ 1 + x_real + x_spur,
     prior = c(prior(normal(0, 10), class = Intercept),
               prior(normal(0, 1), class = b),
               prior(uniform(0, 10), class = sigma)),
     iter = 2000, warmup = 500, chains = 4, cores = 4,
     seed = 5) %>%
fixef() %>%
round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## Intercept	0.00	0.10	-0.19	0.19
## x_real	0.98	0.15	0.69	1.27
## x_spur	0.06	0.09	-0.12	0.24

5.2 Masked relationship

Let's load those tasty milk data.

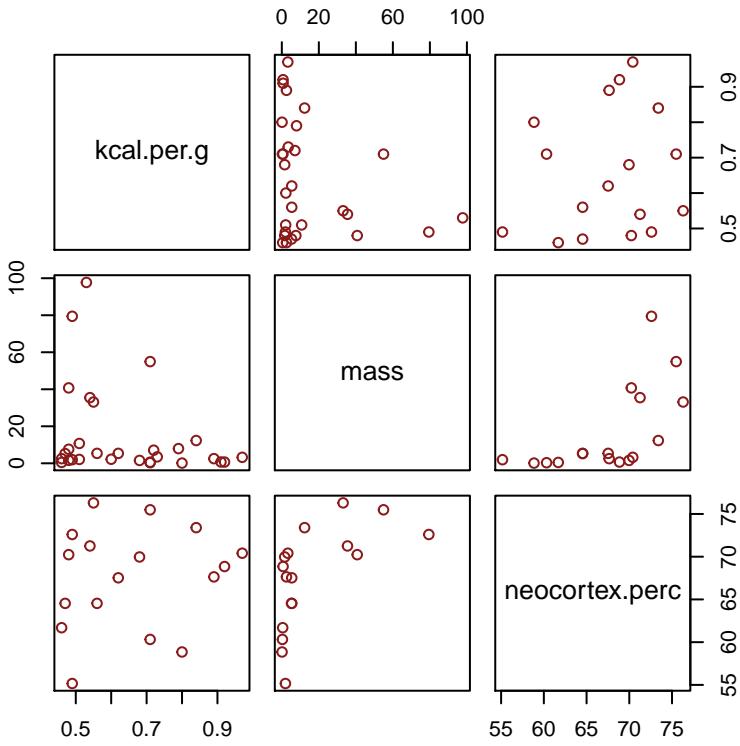
```
library(rethinking)
data(milk)
d <- milk
```

Unload rethinking and load brms.

```
rm(milk)
detach(package:rethinking, unload = T)
library(brms)
```

You might inspect the data like this.

```
d %>%
  select(kcal.per.g, mass, neocortex.perc) %>%
  pairs(col = "firebrick4")
```



By just looking at that mess, do you think you could describe the associations of `mass` and `neocortex.perc` with the criterion, `kcal.per.g`? I couldn't. It's a good thing we have math.

McElreath has us start of with a simple univaraible `milk` model.

```
b5.5 <-
  brm(data = d, family = gaussian,
    kcal.per.g ~ 1 + neocortex.perc,
    prior = c(prior(normal(0, 100), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(cauchy(0, 1), class = sigma)),
    iter = 2000, warmup = 500, chains = 4, cores = 4,
    seed = 5)
```

The uniform prior was difficult on Stan. After playing around a bit, I just switched to a unit-scale half Cauchy. Similar to the rethinking example in the text, brms warned that “Rows containing NAs were excluded from the model.” This isn’t necessarily a problem; the model fit just fine. But we should be ashamed of ourselves and look eagerly forward to Chapter 14 where we’ll learn how to do better.

To compliment how McElreath removed cases with missing values on our variables of interest with Base R `complete.cases()`, here we’ll do so with `tidyverse::drop_na()` and a little help with `ends_with()`.

```
dcc <-
  d %>%
  drop_na(ends_with("_s"))
```

But anyway, let’s inspect the parameter summary.

```
print(b5.5, digits = 3)
```

```
## Family: gaussian
```

```

##   Links: mu = identity; sigma = identity
## Formula: kcal.per.g ~ 1 + neocortex.perc
##   Data: d (Number of observations: 17)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      0.357     0.560   -0.770    1.483      5182 1.002
## neocortex.perc  0.004     0.008   -0.012    0.021      5196 1.002
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       0.192     0.039    0.132    0.283      3532 1.000
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Did you notice now we set `digits = 3` within `print()` much the way McElreath set `digits=3` within `precis()`?

To get the brms answer to what McElreath did with `coef()`, we'll use the `fixef()` function.

```
fixef(b5.5) [2] * (76 - 55)
```

```
## [1] 0.09351993
```

Yes, indeed, “that's less than 0.1 kilocalories” (p. 137).

Just for kicks, we'll superimpose 50% intervals atop 95% intervals for the next few plots. Here's Figure 5.7, top left.

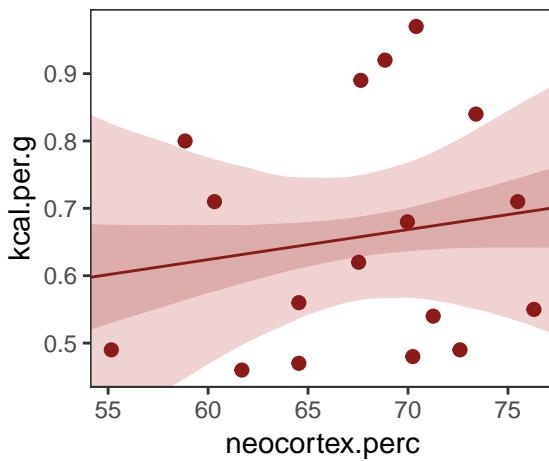
```

nd <- tibble(neocortex.perc = 54:80)

fitted(b5.5,
       newdata = nd,
       probs = c(.025, .975, .25, .75)) %>%
as_tibble() %>%
bind_cols(nd) %>%

ggplot(aes(x = neocortex.perc, y = Estimate)) +
geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5),
            fill = "firebrick", alpha = 1/5) +
geom_smooth(aes(ymin = Q25, ymax = Q75),
            stat = "identity",
            fill = "firebrick4", color = "firebrick4", alpha = 1/5, size = 1/2) +
geom_point(data = nd,
            aes(y = kcal.per.g),
            size = 2, color = "firebrick4") +
coord_cartesian(xlim = range(nd$neocortex.perc),
                 ylim = range(nd$kcal.per.g)) +
ylab("kcal.per.g") +
theme_bw() +
theme(panel.grid = element_blank())

```



Do note the `probs` argument in the `fitted()` code, above. Let's make the `log_mass` variable.

```
dcc <-
  dcc %>%
  mutate(log_mass = log(mass))
```

Now we use `log_mass` as the new sole predictor.

```
b5.6 <-
  brm(data = dcc, family = gaussian,
    kcal.per.g ~ 1 + log_mass,
    prior = c(prior(normal(0, 100), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(uniform(0, 1), class = sigma)),
    iter = 2000, warmup = 500, chains = 4, cores = 4,
    control = list(adapt_delta = 0.9),
    seed = 5)

print(b5.6, digits = 3)

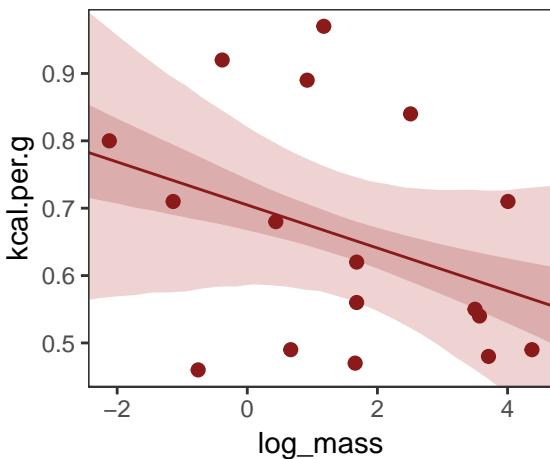
##  Family: gaussian
##  Links: mu = identity; sigma = identity
##  Formula: kcal.per.g ~ 1 + log_mass
##  Data: dcc (Number of observations: 17)
##  Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##            total post-warmup samples = 6000
##
##  Population-Level Effects:
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##  Intercept      0.705     0.060     0.586     0.821       5103 1.000
##  log_mass      -0.032     0.024    -0.080     0.017       4577 1.000
##
##  Family Specific Parameters:
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##  sigma        0.183     0.037     0.128     0.270       3747 1.001
##
##  Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
##  is a crude measure of effective sample size, and Rhat is the potential
##  scale reduction factor on split chains (at convergence, Rhat = 1).
```

Make Figure 5.7, top right.

```
nd <- tibble(log_mass = seq(from = -2.5, to = 5, length.out = 30))

fitted(b5.6,
       newdata = nd,
       probs = c(.025, .975, .25, .75)) %>%
as_tibble() %>%
bind_cols(nd) %>%

ggplot(aes(x = log_mass, y = Estimate)) +
geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5),
            fill = "firebrick", alpha = 1/5) +
geom_smooth(aes(ymin = Q25, ymax = Q75),
            stat = "identity",
            fill = "firebrick4", color = "firebrick4", alpha = 1/5, size = 1/2) +
geom_point(data = dcc,
            aes(y = kcal.per.g),
            size = 2, color = "firebrick4") +
coord_cartesian(xlim = range(dcc$log_mass),
                 ylim = range(dcc$kcal.per.g)) +
ylab("kcal.per.g") +
theme_bw() +
theme(panel.grid = element_blank())
```



Finally, we're ready to fit with both predictors included in the “joint model.” Here's the statistical formula

$$\begin{aligned} \text{kcal.per.g}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta_1 \text{neocortex.perc}_i + \beta_2 \log(\text{mass}_i) \\ \alpha &\sim \text{Normal}(0, 100) \\ \beta_1 &\sim \text{Normal}(0, 1) \\ \beta_2 &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{Uniform}(0, 1) \end{aligned}$$

Note, the HMC chains required a longer `warmup` period and a higher `adapt_delta` setting for the model to converge properly. Life will be much better once we ditch the uniform prior for good.

```
b5.7 <-
brm(data = dcc, family = gaussian,
      kcal.per.g ~ 1 + neocortex.perc + log_mass,
      prior = c(prior(normal(0, 100), class = Intercept),
                prior(normal(0, 1), class = b),
                prior(uniform(0, 1), class = sigma)),
      iter = 4000, warmup = 2000, chains = 4, cores = 4,
```

```

control = list(adapt_delta = 0.999),
seed = 5)

print(b5.7, digits = 3)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: kcal.per.g ~ 1 + neocortex.perc + log_mass
## Data: dcc (Number of observations: 17)
## Samples: 4 chains, each with iter = 4000; warmup = 2000; thin = 1;
##          total post-warmup samples = 8000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      -1.081     0.575   -2.206    0.078       3660 1.000
## neocortex.perc    0.028     0.009    0.010    0.045       3561 1.000
## log_mass        -0.096     0.028   -0.150   -0.040       3424 1.001
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma      0.139     0.030    0.095    0.212       3137 1.002
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Make Figure 5.7, bottom left.

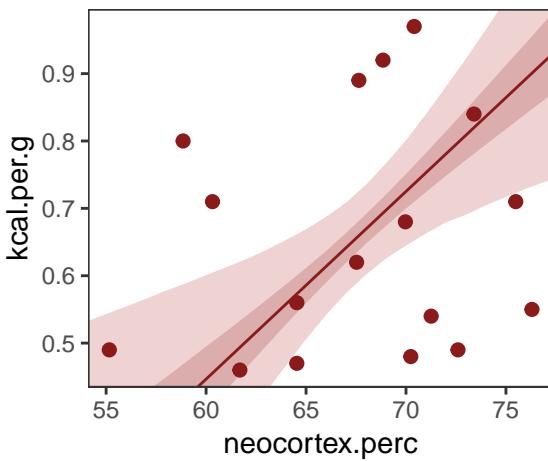
```

nd <-
  tibble(neocortex.perc = 54:80 %>% as.double(),
         log_mass       = mean(dcc$log_mass))

b5.7 %>%
  fitted(newdata = nd,
         probs = c(.025, .975, .25, .75)) %>%
  as_tibble() %>%
  bind_cols(nd) %>%

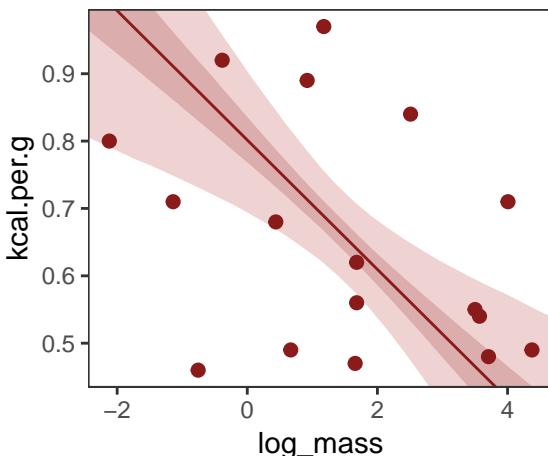
  ggplot(aes(x = neocortex.perc, y = Estimate)) +
  geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5),
              fill = "firebrick", alpha = 1/5) +
  geom_smooth(aes(ymin = Q25, ymax = Q75),
              stat = "identity",
              fill = "firebrick4", color = "firebrick4", alpha = 1/5, size = 1/2) +
  geom_point(data = dcc,
             aes(y = kcal.per.g),
             size = 2, color = "firebrick4") +
  coord_cartesian(xlim = range(dcc$neocortex.perc),
                  ylim = range(dcc$kcal.per.g)) +
  ylab("kcal.per.g") +
  theme_bw() +
  theme(panel.grid = element_blank())

```



And make Figure 5.7, bottom right.

```
nd <-  
  tibble(log_mass      = seq(from = -2.5, to = 5, length.out = 30),  
         neocortex.perc = mean(dcc$neocortex.perc))  
  
b5.7 %>%  
  fitted(newdata = nd,  
         probs = c(.025, .975, .25, .75)) %>%  
  as_tibble() %>%  
  bind_cols(nd) %>%  
  
  ggplot(aes(x = log_mass, y = Estimate)) +  
  geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5),  
              fill = "firebrick", alpha = 1/5) +  
  geom_smooth(aes(ymin = Q25, ymax = Q75),  
              stat = "identity",  
              fill = "firebrick4", color = "firebrick4", alpha = 1/5, size = 1/2) +  
  geom_point(data = dcc,  
             aes(y = kcal.per.g),  
             size = 2, color = "firebrick4") +  
  coord_cartesian(xlim = range(dcc$log_mass),  
                  ylim = range(dcc$kcal.per.g)) +  
  ylab("kcal.per.g") +  
  theme_bw() +  
  theme(panel.grid = element_blank())
```



What [this regression model did was] ask if species that have high neocortex percent *for their body mass* have higher milk energy. Likewise, the model [asked] if species with high body mass *for their neocortex percent* have

higher milk energy. Bigger species, like apes, have milk with less energy. But species with more neocortex tend to have richer milk. The fact that these two variables, body size and neocortex, are correlated across species makes it hard to see these relationships, unless we statistically account for both. (pp. 140–141, *emphasis* in the original)

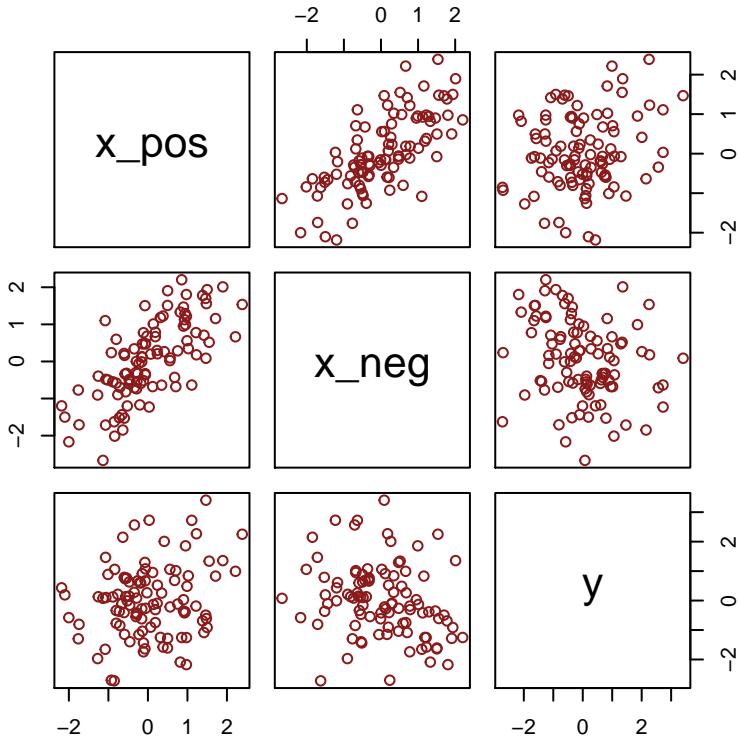
5.2.0.1 Overthinking: Simulating a masking relationship.

```
n      <- 100    # number of cases
rho    <- .7     # correlation between x_pos and x_neg

set.seed(5)  # setting the seed makes the results reproducible
d <-
  tibble(x_pos = rnorm(n),                      # x_pos as a standard Gaussian
         x_neg = rnorm(n, rho * x_pos, sqrt(1 - rho^2)),  # x_neg correlated with x_pos
         y      = rnorm(n, x_pos - x_neg))                # y equally associated with x_pos and x_neg
```

Here are the quick `pairs()` plots.

```
pairs(d, col = "firebrick4")
```



Here we fit the models with a little help from the `update()` function.

```
b5.0.both <-
  brm(data = d, family = gaussian,
       y ~ 1 + x_pos + x_neg,
       prior = c(prior(normal(0, 100), class = Intercept),
                 prior(normal(0, 1), class = b),
                 prior(cauchy(0, 1), class = sigma)),
       seed = 5)

b5.0.pos <-
  update(b5.0.both,
         formula = y ~ 1 + x_pos)

b5.0.neg <-
```

```
update(b5.0.both,
       formula = y ~ 1 + x_neg)
```

Compare the coefficients.

```
fixef(b5.0.pos) %>% round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## Intercept	-0.01	0.12	-0.24	0.23
## x_pos	0.26	0.13	0.01	0.51

```
fixef(b5.0.neg) %>% round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## Intercept	0.01	0.12	-0.23	0.24
## x_neg	-0.29	0.11	-0.50	-0.06

```
fixef(b5.0.both) %>% round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## Intercept	0.00	0.10	-0.20	0.19
## x_pos	0.97	0.15	0.67	1.24
## x_neg	-0.90	0.13	-1.15	-0.64

5.3 Multicollinearity

Multicollinearity means very strong correlation between two or more predictor variables. The consequence of it is that the posterior distribution will say that a very large range of parameter values are plausible, from tiny associations to massive ones, even if all of the variables are in reality strongly associated with the outcome. This frustrating phenomenon arises from the details of how statistical control works. So once you understand multicollinearity, you will better understand [multivariable] models in general. (pp. 141–142)

5.3.1 Multicollinear legs.

Let's simulate some leg data.

```
n <- 100
set.seed(5)

d <-
  tibble(height      = rnorm(n, mean = 10, sd = 2),
         leg_prop   = runif(n, min = 0.4, max = 0.5)) %>%
  mutate(leg_left  = leg_prop * height + rnorm(n, mean = 0, sd = 0.02),
        leg_right = leg_prop * height + rnorm(n, mean = 0, sd = 0.02))
```

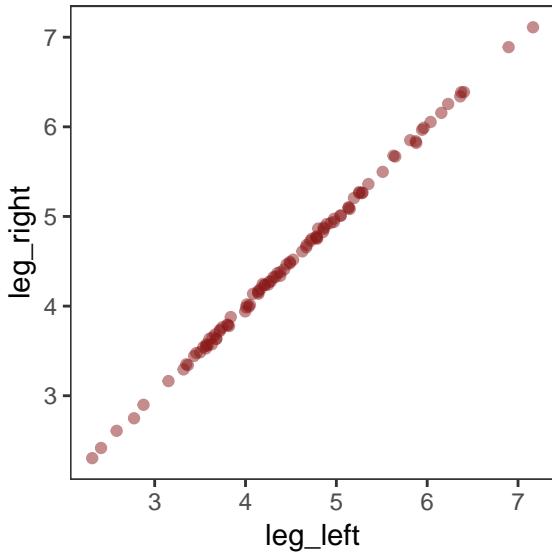
leg_left and leg_right are highly correlated.

```
d %>%
  select(leg_left:leg_right) %>%
  cor() %>%
  round(digits = 4)
```

	leg_left	leg_right
## leg_left	1.0000	0.9996
## leg_right	0.9996	1.0000

Have you ever even seen a $\rho = .9996$ correlation, before? Here it is in a plot.

```
d %>%
  ggplot(aes(x = leg_left, y = leg_right)) +
  geom_point(alpha = 1/2, color = "firebrick4") +
  theme_bw() +
  theme(panel.grid = element_blank())
```



Here's our attempt to predict height with both legs.

```
b5.8 <-
  brm(data = d, family = gaussian,
    height ~ 1 + leg_left + leg_right,
    prior = c(prior(normal(10, 100), class = Intercept),
              prior(normal(2, 10), class = b),
              prior(uniform(0, 10), class = sigma)),
    iter = 2000, warmup = 500, chains = 4, cores = 4,
    seed = 5)
```

Let's inspect the damage.

```
print(b5.8)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 1 + leg_left + leg_right
##   Data: d (Number of observations: 100)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##           total post-warmup samples = 6000
##
## Population-Level Effects:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      1.79      0.29     1.21     2.35      5680  1.00
## leg_left       0.72      2.16    -3.45     5.04      1808  1.00
## leg_right      1.12      2.16    -3.19     5.29      1810  1.00
##
## Family Specific Parameters:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma        0.61      0.04     0.53     0.70      3593  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
```

```
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

That ‘Est.Error’ column isn’t looking too good. But it’s easy to miss that, which is why McElreath suggested “a graphical view of the [output] is more useful because it displays the posterior [estimates] and [intervals] in a way that allows us with a glance to see that something has gone wrong here” (p. 143).

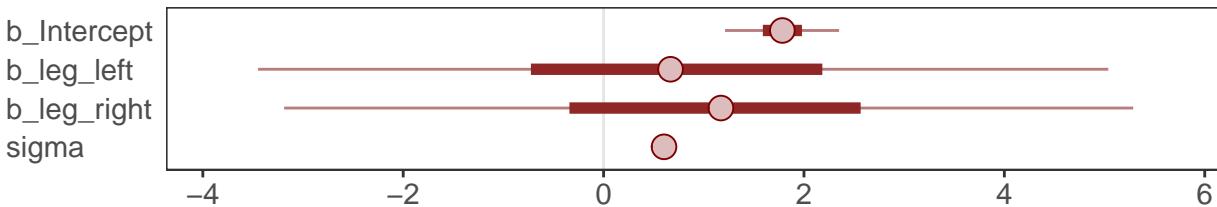
Here’s our coefficient plot using `brms::stanplot()` with a little help from `bayesplot::color_scheme_set()`.

```
color_scheme_set("red")

stanplot(b5.8,
         type = "intervals",
         prob = .5,
         prob_outer = .95,
         point_est = "median") +
  labs(title    = "The coefficient plot for the two-leg model",
       subtitle = "Holy smokes; look at the widths of those betas!") +
  theme_bw() +
  theme(text      = element_text(size = 14),
        panel.grid = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0))
```

The coefficient plot for the two-leg model

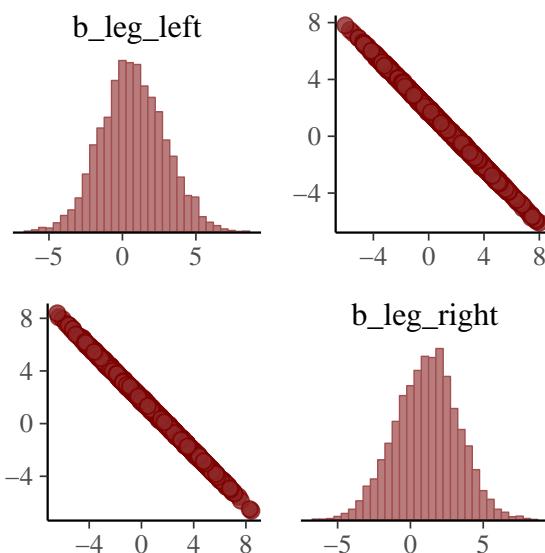
Holy smokes; look at the widths of those betas!



Now you can use the `brms::stanplot()` function without explicitly loading the `bayesplot` package. But loading `bayesplot` allows you to set the color scheme with `color_scheme_set()`.

This is perhaps the simplest way to plot the bivariate posterior of our two predictor coefficients, Figure 6.2.a.

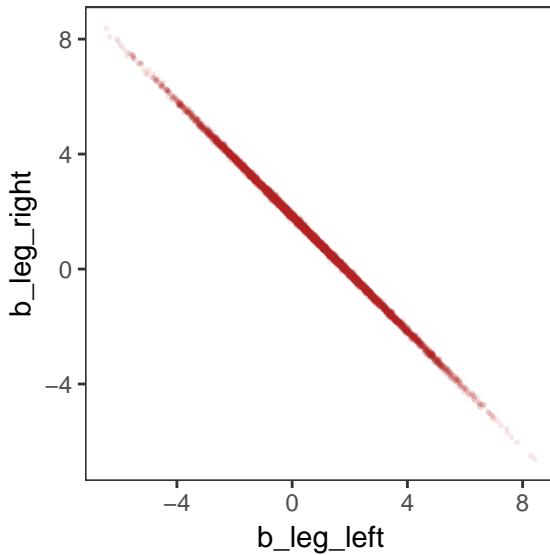
```
pairs(b5.8, pars = parnames(b5.8)[2:3])
```



If you’d like a nicer and more focused attempt, you might have to revert to the `posterior_samples()` function and a little `ggplot2` code.

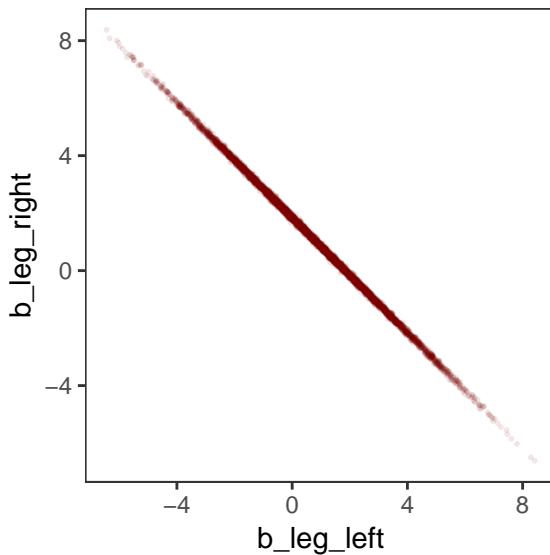
```
post <- posterior_samples(b5.8)

post %>%
  ggplot(aes(x = b_leg_left, y = b_leg_right)) +
  geom_point(color = "firebrick", alpha = 1/10, size = 1/3) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



While we're at it, you can make a similar plot with the `mcmc_scatter()` function.

```
post %>%
  mcmc_scatter(pars = c("b_leg_left", "b_leg_right"),
                size = 1/3,
                alpha = 1/10) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



But wow, those coefficients look about as highly correlated as the predictors, just with the reversed sign.

```
post %>%
  select(b_leg_left:b_leg_right) %>%
  cor()
```

```
##           b_leg_left b_leg_right
## b_leg_left    1.0000000 -0.9995795
## b_leg_right   -0.9995795  1.0000000
```

On page 165, McElreath clarified that “from the computer’s perspective, this model is simply:”

$$\begin{aligned}y_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + (\beta_1 + \beta_2)x_i\end{aligned}$$

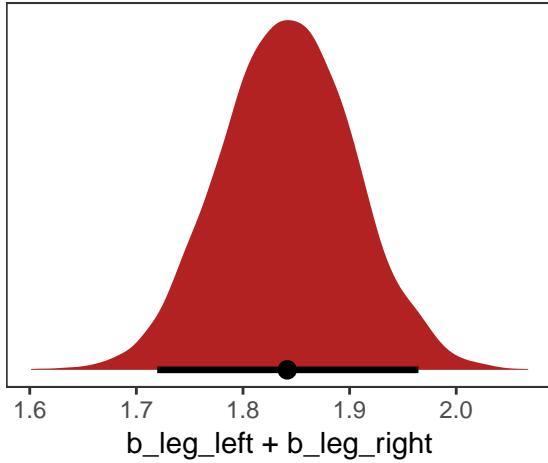
Accordingly, here’s the posterior of the sum of the two regression coefficients, Figure 6.2.b. We’ll use `tidybayes::geom_halfeyeh()` to both plot the density and mark off the posterior median and percentile-based 95% probability intervals at its base.

```
library(tidybayes)

post %>%
  ggplot(aes(x = b_leg_left + b_leg_right, y = 0)) +
  geom_halfeyeh(fill = "firebrick",
                 point_interval = median_qi, .width = .95) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(title = "Sum the multicollinear coefficients",
       subtitle = "Marked by the median and 95% PIs") +
  theme_bw() +
  theme(panel.grid = element_blank())
```

Sum the multicollinear coefficients

Marked by the median and 95% PIs



Now we fit the model after ditching one of the leg lengths.

```
b5.9 <-
  brm(data = d, family = gaussian,
       height ~ 1 + leg_left,
       prior = c(prior(normal(10, 100), class = Intercept),
                 prior(normal(2, 10), class = b),
                 prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

```
print(b5.9)

## Family: gaussian
##   Links: mu = identity; sigma = identity
```

```

## Formula: height ~ 1 + leg_left
##   Data: d (Number of observations: 100)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##           total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      1.79      0.28     1.23     2.34        7237 1.00
## leg_left       1.84      0.06     1.72     1.96        7355 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma         0.60      0.04     0.53     0.69        7203 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

That posterior SD looks much better. Compare this density to the one in Figure 6.1.b.

```
posterior_samples(b5.9) %>%
```

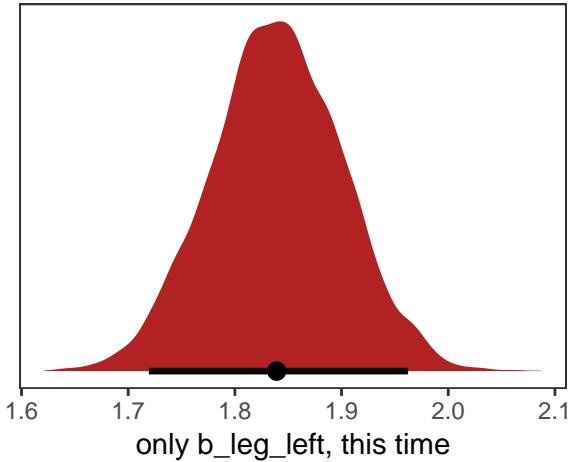
```

ggplot(aes(x = b_leg_left, y = 0)) +
  geom_halfeyeh(fill = "firebrick",
                 point_interval = median_qi, .width = .95) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(title = "Just one coefficient needed",
       subtitle = "Marked by the median and 95% PIs",
       x = "only b_leg_left, this time") +
  theme_bw() +
  theme(panel.grid = element_blank())

```

Just one coefficient needed

Marked by the median and 95% PIs



When two predictor variables are very strongly correlated, including both in a model may lead to confusion. The posterior distribution isn't wrong, in such a case. It's telling you that the question you asked cannot be answered with these data. And that's a great thing for a model to say, that it cannot answer your question. (p. 145, *emphasis* in the original)

5.3.2 Multicollinear milk.

Multicollinearity arises in real data, too.

```
library(rethinking)
data(milk)
d <- milk
```

Unload rethinking and load brms.

```
rm(milk)
detach(package:rethinking, unload = TRUE)
library(brms)
```

We'll follow the text and fit the two univariable models, first. Note our use of the `update()` function.

```
# `kcal.per.g` regressed on `perc.fat`
b5.10 <-
  brm(data = d, family = gaussian,
    kcal.per.g ~ 1 + perc.fat,
    prior = c(prior(normal(.6, 10), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(uniform(0, 10), class = sigma)),
    iter = 2000, warmup = 500, chains = 4, cores = 4,
    seed = 5)

# `kcal.per.g` regressed on `perc.lactose`
b5.11 <-
  update(b5.10,
    newdata = d,
    formula = kcal.per.g ~ 1 + perc.lactose)
```

Compare the coefficients.

```
posterior_summary(b5.10) %>% round(digits = 3)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	0.301	0.040	0.223	0.379
## b_perc.fat	0.010	0.001	0.008	0.012
## sigma	0.080	0.011	0.061	0.106
## lp__	24.017	1.275	20.740	25.490

```
posterior_summary(b5.11) %>% round(digits = 3)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	1.167	0.047	1.078	1.261
## b_perc.lactose	-0.011	0.001	-0.012	-0.009
## sigma	0.067	0.010	0.051	0.090
## lp__	28.776	1.273	25.582	30.273

If you'd like to get just the 95% intervals similar to the way McElreath reported them in the prose on page 146, you might use the handy `posterior_interval()` function.

```
posterior_interval(b5.10)[2, ] %>% round(digits = 3)
```

```
## 2.5% 97.5%
## 0.008 0.012
```

```
posterior_interval(b5.11)[2, ] %>% round(digits = 3)
```

```
##    2.5% 97.5%
## -0.012 -0.009
```

Now “watch what happens when we place both predictor variables in the same regression model” (p. 146).

```
b5.12 <-
  update(b5.11,
         newdata = d,
         formula = kcal.per.g ~ 1 + perc.fat + perc.lactose)
```

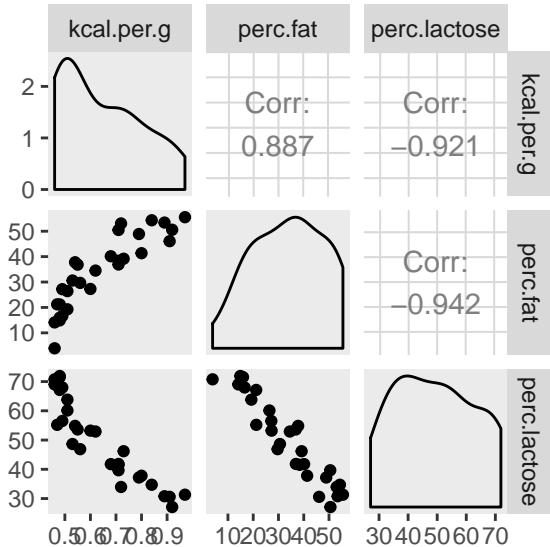
```
posterior_summary(b5.12) %>% round(digits = 3)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	1.008	0.227	0.558	1.455
## b_perc.fat	0.002	0.003	-0.003	0.007
## b_perc.lactose	-0.009	0.003	-0.014	-0.003
## sigma	0.068	0.010	0.051	0.091
## lp_	27.570	1.535	23.649	29.514

You can make custom pairs plots with [GGally](#), which will also compute the point estimates for the bivariate correlations. Here's a default plot.

```
#install.packages("GGally", dependencies = T)
library(GGally)

ggpairs(data = d, columns = c(3:4, 6)) +
  theme(panel.grid = element_blank())
```



But you can customize [these](#), too. E.g.,

```
my_diag <- function(data, mapping, ...){
  ggplot(data = data, mapping = mapping) +
  geom_density(fill = "firebrick4", size = 0)
}

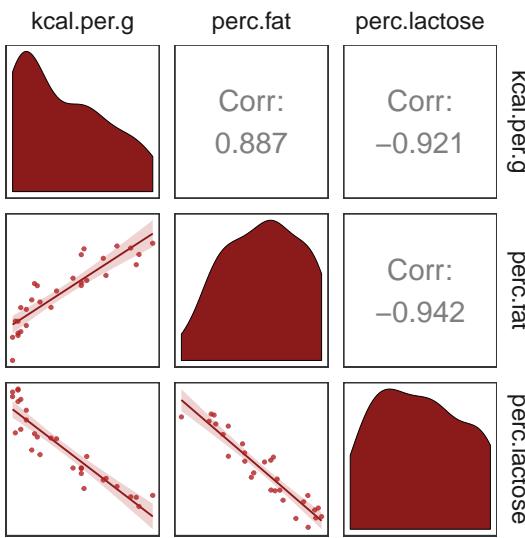
my_lower <- function(data, mapping, ...){
  ggplot(data = data, mapping = mapping) +
```

```

geom_smooth(method = "lm", color = "firebrick4", size = 1/3,
            fill = "firebrick", alpha = 1/5) +
geom_point(color = "firebrick", alpha = .8, size = 1/4)
}

# then plug those custom functions into `ggpairs()`
ggpairs(data = d, columns = c(3:4, 6),
        diag = list(continuous = my_diag),
        lower = list(continuous = my_lower)) +
theme_bw() +
theme(strip.background = element_rect(fill = "white", color = "white"),
      axis.text = element_blank(),
      axis.ticks = element_blank(),
      panel.grid = element_blank())

```



Our two predictor “variables are negatively correlated, and so strongly so that they are nearly redundant. Either helps in predicting `kcal.per.g`, but neither helps much *once you already know the other*” (p. 148, *emphasis* in the original). You can really see that on the lower two scatter plots. You’ll note the `ggpairs()` plot also showed the Pearson’s correlation coefficients, so we don’t need to use the `cor()` function like McElreath did in the text.

In the next section, we’ll run the simulation necessary for our version of Figure 5.10.

5.3.2.1 Overthinking: Simulating collinearity.

First we’ll get the data and define the functions. You’ll note I’ve defined my `sim_coll()` a little differently from `sim.coll()` in the text. I’ve omitted `rep.sim.coll()` as an independent function altogether, but computed similar summary information with the `summarise()` code at the bottom of the block.

```

sim_coll <- function(seed, rho){
  set.seed(seed)
  d <-
  d %>%
  mutate(x = rnorm(n(),
                  mean = perc.fat * rho,
                  sd   = sqrt((1 - rho^2) * var(perc.fat))))
  m <- lm(kcal.per.g ~ perc.fat + x, data = d)
  sqrt(diag(vcov(m)))[2] # parameter SD
}

# how many simulations per `rho`-value would you like?

```

```

n_seed <- 100
# how many `rho`-values from 0 to .99 would you like to evaluate the process over?
n_rho <- 30

d <-
  tibble(seed = 1:n_seed) %>%
  expand(seed, rho = seq(from = 0, to = .99, length.out = n_rho)) %>%
  mutate(parameter_sd = purrr::map2_dbl(seed, rho, sim_coll)) %>%
  group_by(rho) %>%
  # we'll `summarise()` our output by the mean and 95% intervals
  summarise(mean = mean(parameter_sd),
            ll   = quantile(parameter_sd, prob = .025),
            ul   = quantile(parameter_sd, prob = .975))

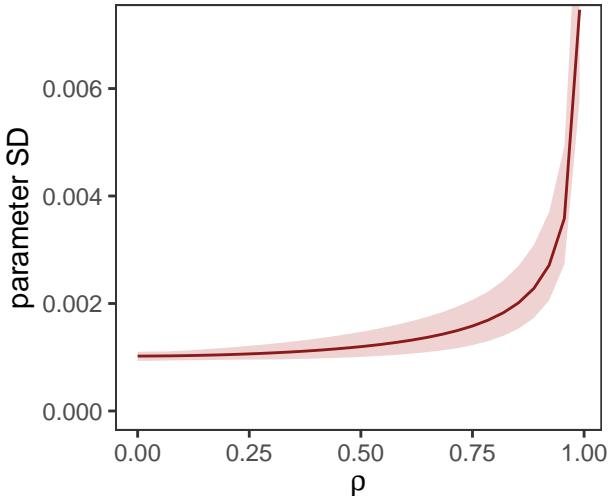
```

We've added 95% interval bands to our version of Figure 5.10.

```

d %>%
  ggplot(aes(x = rho, y = mean)) +
  geom_smooth(aes(ymin = ll, ymax = ul),
              stat = "identity",
              fill = "firebrick", color = "firebrick4", alpha = 1/5, size = 1/2) +
  labs(x = expression(rho),
       y = "parameter SD") +
  coord_cartesian(ylim = c(0, .0072)) +
  theme_bw() +
  theme(panel.grid = element_blank())

```



Did you notice we used the base R `lm()` function to fit the models? As McElreath rightly pointed out, `lm()` presumes flat priors. Proper Bayesian modeling could improve on that. But then we'd have to wait for a whole lot of HMC chains to run and until our personal computers or the algorithms we use to fit our Bayesian models become orders of magnitude faster, we just don't have time for that.

5.3.3 Post-treatment bias.

It helped me understand the next example by mapping out the sequence of events McElreath described in the second paragraph:

- seed and sprout plants
- measure heights
- apply different antifungal soil treatments (i.e., the experimental manipulation)
- measure (a) the heights and (b) the presence of fungus

Based on the design, let's simulate our data.

```
n <- 100

set.seed(5)
d <-
  tibble(h0      = rnorm(n, mean = 10, sd = 2),
         treatment = rep(0:1, each = n / 2),
         fungus    = rbinom(n, size = 1, prob = .5 - treatment * 0.4),
         h1        = h0 + rnorm(n, mean = 5 - 3 * fungus, sd = 1))
```

We'll use `head()` to peek at the data.

```
d %>%
  head()

## # A tibble: 6 x 4
##   h0 treatment fungus   h1
##   <dbl>     <int> <int> <dbl>
## 1 8.32       0     0 14.3
## 2 12.8       0     0 18.5
## 3 7.49       0     1  8.97
## 4 10.1       0     1 12.9
## 5 13.4       0     1 14.8
## 6 8.79       0     1 12.0
```

These data + the model were rough on Stan, at first, which spat out warnings about divergent transitions. The model ran fine after setting `warmup = 1000` and `adapt_delta = 0.99`.

```
b5.13 <-
  brm(data = d, family = gaussian,
       h1 ~ 1 + h0 + treatment + fungus,
       prior = c(prior(normal(0, 100), class = Intercept),
                  prior(normal(0, 10), class = b),
                  prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = 0.99),
       seed = 5)

print(b5.13)

## Family: gaussian
##  Links: mu = identity; sigma = identity
## Formula: h1 ~ 1 + h0 + treatment + fungus
## Data: d (Number of observations: 100)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     5.58      0.56    4.49    6.69      2954 1.00
## h0            0.94      0.05    0.83    1.05      3313 1.00
## treatment     0.05      0.22   -0.41    0.48      917  1.00
## fungus       -2.76      0.26   -3.26   -2.26      1950 1.00
##
## Family Specific Parameters:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma         1.01      0.08    0.87    1.16      3046 1.00
```

```
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Now fit the model after excluding `fungus`, our post-treatment variable.

```
b5.14 <-  
  update(b5.13,  
    formula = h1 ~ 1 + h0 + treatment)
```

```
print(b5.14)
```

```
## Family: gaussian  
##   Links: mu = identity; sigma = identity  
## Formula: h1 ~ h0 + treatment  
##   Data: d (Number of observations: 100)  
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;  
##           total post-warmup samples = 4000  
##  
## Population-Level Effects:  
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## Intercept      5.49     0.83     3.87     7.13      1882 1.00  
## h0              0.82     0.08     0.67     0.98      1956 1.00  
## treatment       1.01     0.31     0.39     1.60      1748 1.00  
##  
## Family Specific Parameters:  
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## sigma          1.53     0.12     1.32     1.76      1153 1.00  
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

“Now the impact of treatment is strong and positive, as it should be” (p. 152). In this case, there were really two outcomes. The first was the one we modeled, the height at the end of the experiment (i.e., `h1`). The second outcome, which was clearly related to `h1`, was the presence of fungus, captured by our binomial variable `fungus`. If you wanted to model that, you’d fit a logistic regression model, which we’ll learn about in Chapter 10.

5.4 Categorical variables

Many readers will already know that variables like this, routinely called *factors*, can easily be included in linear models. But what is not widely understood is how these variables are included in a model... Knowing how the machine works removes a lot of this difficulty. (p. 153, *emphasis* in the original)

5.4.1 Binary categories.

Reload the `Howell1` data.

```
library(rethinking)  
data(Howell1)  
d <- Howell1
```

Unload `rethinking` and load `brms`.

```
rm(Howell1)
detach(package:rstan, unload = T)
library(brms)
```

Just in case you forgot what these data were like:

```
d %>%
  glimpse()
```

```
## Observations: 544
## Variables: 4
## $ height <dbl> 151.7650, 139.7000, 136.5250, 156.8450, 145.4150, 163.8...
## $ weight <dbl> 47.82561, 36.48581, 31.86484, 53.04191, 41.27687, 62.99...
## $ age     <dbl> 63.0, 63.0, 65.0, 41.0, 51.0, 35.0, 32.0, 27.0, 19.0, 5...
## $ male    <int> 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1...
```

Let's fit the first `height` model with the `male` dummy.

Note. The uniform prior McElreath used in the text in conjunction with the `brms::brm()` function seemed to cause problems for the HMC chains, here. After experimenting with start values, increasing `warmup`, and increasing `adapt_delta`, switching out the uniform prior did the trick. Anticipating Chapter 8, I recommend you use a weakly-regularizing half Cauchy for σ .

```
b5.15 <-
  brm(data = d, family = gaussian,
       height ~ 1 + male,
       prior = c(prior(normal(178, 100), class = Intercept),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 2), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

Check the summary.

```
print(b5.15)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 1 + male
## Data: d (Number of observations: 544)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##           total post-warmup samples = 6000
##
## Population-Level Effects:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept    134.84      1.61    131.63    137.99      6321  1.00
## male         7.23       2.31     2.66     11.84      6303  1.00
##
## Family Specific Parameters:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma        27.38      0.84    25.80    29.06      6445  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Our samples from the posterior are already in the HMC iterations. All we need to do is put them in a data frame and then put them to work.

```
post <- posterior_samples(b5.15)

post %>%
  transmute(male_height = b_Intercept + b_male) %>%
  mean_qi(.width = .89)

##   male_height    .lower   .upper .width .point .interval
## 1    142.0679 139.3071 144.855    0.89    mean       qi
```

You can also do this with `fitted()`.

```
nd <- tibble(male = 1)

fitted(b5.15,
       newdata = nd)

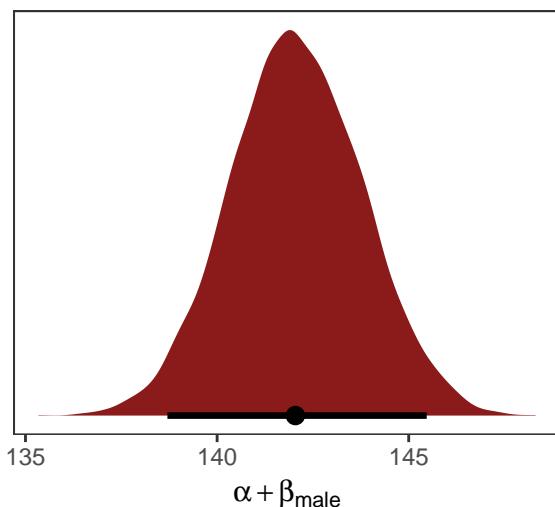
##      Estimate Est.Error    Q2.5    Q97.5
## [1,] 142.0679 1.723907 138.7037 145.4691
```

And you could even plot.

```
fitted(b5.15,
       newdata = nd,
       summary = F) %>%
  as_tibble() %>%

ggplot(aes(x = V1, y = 0)) +
  geom_halfeyeh(fill = "firebrick4",
                 point_interval = median_qi, .width = .95) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(subtitle = "Model-implied male heights",
       x = expression(alpha + beta["male"])) +
  theme_bw() +
  theme(panel.grid = element_blank())
```

Model-implied male heights



5.4.1.1 Overthinking: Re-parameterizing the model.

The reparameterized model follows the form

$$\begin{aligned} \text{height}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha_{\text{female}}(1 - \text{male}_i) + \alpha_{\text{male}}\text{male}_i \end{aligned}$$

So then a `female` dummy would satisfy the condition $\text{female}_i = (1 - \text{male}_i)$. Let's make that dummy.

```
d <-
d %>%
  mutate(female = 1 - male)
```

Everyone has their own idiosyncratic way of coding. One of my quirks is I always explicitly specify a model's intercept following the form $y \sim 1 + x$, where y is the criterion, x stands for the predictors, and 1 is the intercept. You don't have to do this, of course. You could just code $y \sim x$ to get the same results. The `brm()` function assumes you want that intercept. One of the reasons I like the verbose version is it reminds me to think about the intercept and to include it in my priors. Another nice feature is that it helps me make sense of the code for this model: `height ~ 0 + male + female`. When we replace $\dots \sim 1 + \dots$ with $\dots \sim 0 + \dots$, we tell `brm()` to remove the intercept. Removing the intercept allows us to include ALL levels of a given categorical variable in our model. In this case, we've expressed sex as two dummies, `female` and `male`. Taking out the intercept lets us put both dummies into the formula.

```
b5.15b <-
  brm(data = d, family = gaussian,
       height ~ 0 + male + female,
       prior = c(prior(normal(178, 100), class = b),
                 prior(cauchy(0, 2), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)

print(b5.15b)

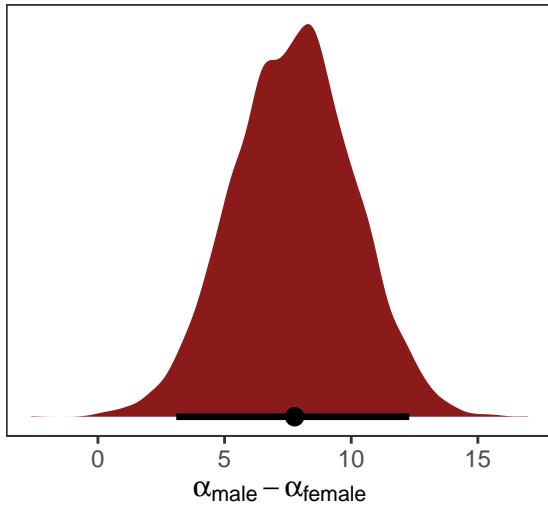
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: height ~ 0 + male + female
##   Data: d (Number of observations: 544)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## male        142.35     1.70   139.04   145.66      5863 1.00
## female      134.63     1.61   131.48   137.70      5442 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       27.38     0.84    25.79    29.12      5000 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

If we wanted the formal difference score from such a model, we'd subtract.

```
posterior_samples(b5.15b) %>%
  transmute(dif = b_male - b_female) %>%
  ggplot(aes(x = dif, y = 0)) +
  geom_halfeyeh(fill = "firebrick4",
                point_interval = median_qi, .width = .95) +
```

```
scale_y_continuous(NULL, breaks = NULL) +
  labs(subtitle = "Model-implied difference score",
       x = expression(alpha["male"] - alpha["female"])) +
  theme_bw() +
  theme(panel.grid = element_blank())
```

Model-implied difference score



5.4.2 Many categories.

When there are more than two categories, you'll need more than one dummy variable. Here's the general rule: To include k categories in a linear model, you require $k - 1$ dummy variables. Each dummy variable indicates, with the value 1, a unique category. The category with no dummy variable assigned to it ends up again as the “intercept” category. (p. 155)

We'll practice with `milk`.

```
library(rethinking)
data(milk)
d <- milk
```

Unload `rethinking` and load `brms`.

```
rm(milk)
detach(package:rethinking, unload = T)
library(brms)
```

With the `tidyverse`, we can peek at `clade` with `distinct()` in the place of base R `unique()`.

```
d %>%
  distinct(clade)

##          clade
## 1      Strepsirrhine
## 2 New World Monkey
## 3 Old World Monkey
## 4          Ape
```

As `clade` has 4 categories, let's use `ifelse()` to convert these to 4 dummy variables.

```
d <-
d %>%
  mutate(clade_nwm = ifelse(clade == "New World Monkey", 1, 0),
         clade_owm = ifelse(clade == "Old World Monkey", 1, 0),
         clade_s   = ifelse(clade == "Strepsirrhine", 1, 0),
         clade_ape = ifelse(clade == "Ape", 1, 0))
```

Now we'll fit the model with three of the four dummies. In this model, `clade_ape` is the reference category captured by the intercept.

```
b5.16 <-
  brm(data = d, family = gaussian,
       kcal.per.g ~ 1 + clade_nwm + clade_owm + clade_s,
       prior = c(prior(normal(.6, 10), class = Intercept),
                 prior(normal(0, 1), class = b),
                 prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

```
print(b5.16)
```

```
## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: kcal.per.g ~ 1 + clade_nwm + clade_owm + clade_s
## Data: d (Number of observations: 29)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     0.55      0.04    0.46    0.63        4771 1.00
## clade_nwm     0.17      0.06    0.05    0.29        4874 1.00
## clade_owm     0.24      0.07    0.11    0.38        5031 1.00
## clade_s      -0.04      0.07   -0.18    0.11        5308 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       0.13      0.02    0.10    0.17        5397 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Here we grab the chains, our draws from the posterior.

```
post <-
  b5.16 %>%
  posterior_samples()

head(post)

##   b_Intercept b_clade_nwm b_clade_owm   b_clade_s     sigma     lp__
## 1  0.5575386  0.1932337  0.2276990  0.043159795 0.1275090 9.271587
## 2  0.4800805  0.2532488  0.2068195  0.010154008 0.1325052 7.787578
## 3  0.5449358  0.1740914  0.3652368  0.006283387 0.1182335 7.644787
## 4  0.4359255  0.2517067  0.3219673  0.128277164 0.1410273 6.847354
## 5  0.5666455  0.2048127  0.2708950 -0.025282543 0.1228911 9.358463
## 6  0.5470347  0.2507482  0.1803077 -0.098373317 0.1195493 7.604434
```

You might compute averages for each category and summarizing the results with the transpose of base R's `apply()` function, rounding to two digits of precision.

```
post$mu_ape <- post$b_Intercept
post$mu_nwm <- post$b_Intercept + post$b_clade_nwm
post$mu_owm <- post$b_Intercept + post$b_clade_owm
post$mu_s   <- post$b_Intercept + post$b_clade_s

round(t(apply(post[ ,7:10], 2, quantile, c(.5, .025, .975))), digits = 2)

##      50% 2.5% 97.5%
## mu_ape 0.55 0.46 0.63
## mu_nwm 0.71 0.63 0.80
## mu_owm 0.79 0.68 0.89
## mu_s   0.51 0.39 0.63
```

Here's a more tidyverse sort of way to get the same thing, but this time with means and HPDIs via the `tidybayes::mean_hdi()` function.

```
post %>%
  transmute(mu_ape = b_Intercept,
            mu_nwm = b_Intercept + b_clade_nwm,
            mu_owm = b_Intercept + b_clade_owm,
            mu_s   = b_Intercept + b_clade_s) %>%
  gather() %>%
  group_by(key) %>%
  mean_hdi() %>%
  mutate_if(is.double, round, digits = 2)
```

```
## # A tibble: 4 x 7
##   key     value .lower .upper .width .point .interval
##   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <chr>   <chr>
## 1 mu_ape  0.55    0.46    0.63    0.95  mean   hdi
## 2 mu_nwm  0.71    0.62    0.8     0.95  mean   hdi
## 3 mu_owm  0.79    0.68    0.89    0.95  mean   hdi
## 4 mu_s    0.51    0.4     0.63    0.95  mean   hdi
```

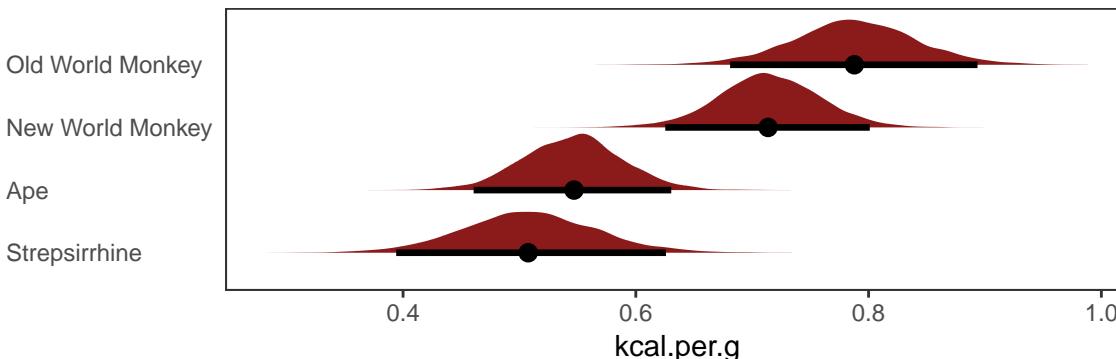
You could also summarize with `fitted()`.

```
nd <- tibble(clade_nwm = c(1, 0, 0, 0),
             clade_owm = c(0, 1, 0, 0),
             clade_s   = c(0, 0, 1, 0),
             primate   = c("New World Monkey", "Old World Monkey", "Strepsirrhine", "Ape"))

fitted(b5.16,
       newdata = nd,
       summary = F) %>%
  as_tibble() %>%
  gather() %>%
  mutate(primate = rep(c("New World Monkey", "Old World Monkey", "Strepsirrhine", "Ape"),
                       each = n() / 4)) %>%

ggplot(aes(x = value, y = reorder(primate, value))) +
  geom_halfeyeh(fill = "firebrick4",
                point_interval = median_qi, .width = .95) +
  labs(x = "kcal.per.g",
       y = NULL) +
  theme_bw() +
```

```
theme(panel.grid    = element_blank(),
      axis.ticks.y = element_blank(),
      axis.text.y  = element_text(hjust = 0))
```



And there are multiple ways to compute summary statistics for the difference between NWM and OWM, too.

```
# base R
quantile(post$mu_nwm - post$mu_owm, probs = c(.5, .025, .975))
```

```
##           50%       2.5%      97.5%
## -0.07362279 -0.21514058  0.06410530
```

```
# tidyverse + tidybayes
post %>%
  transmute(dif = mu_nwm - mu_owm) %>%
  median_qi()
```

```
##          dif     .lower     .upper .width .point .interval
## 1 -0.07362279 -0.2151406  0.0641053   0.95 median      qi
```

5.4.3 Adding regular predictor variables.

If we wanted to fit the model including `perc.fat` as an additional predictor, the basic statistical formula would be

$$\mu_i = \alpha + \beta_{\text{clade_nwm}} \text{clade_nwm}_i + \beta_{\text{clade_owm}} \text{clade_owm}_i + \beta_{\text{clade_s}} \text{clade_s}_i + \beta_{\text{perc.fat}} \text{perc.fat}_i$$

The corresponding `formula` argument within `brm()` would be `kcal.per.g ~ 1 + clade_nwm + clade_owm + clade_s + perc.fat`.

5.4.4 Another approach: Unique intercepts.

Using the code below, there's no need to transform `d$clade` into `d$clade_id`. The advantage of this approach is the indices in the model summary are more descriptive than `a[1]` through `a[4]`.

```
b5.16_alt <-
  brm(data = d, family = gaussian,
       kcal.per.g ~ 0 + clade,
       prior = c(prior(normal(.6, 10), class = b),
                 prior(uniform(0, 10), class = sigma)),
       iter = 2000, warmup = 500, chains = 4, cores = 4,
       seed = 5)
```

```
print(b5.16_alt)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: kcal.per.g ~ 0 + clade
## Data: d (Number of observations: 29)
## Samples: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## cladeApe        0.55     0.04    0.46    0.64      7023 1.00
## cladeNewWorldMonkey   0.71     0.04    0.63    0.80      7420 1.00
## cladeOldWorldMonkey   0.79     0.05    0.68    0.89      6222 1.00
## cladeStrepsirrhine   0.51     0.06    0.39    0.63      7233 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       0.13     0.02    0.10    0.17      4777 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

See? This is much easier than trying to remember which one was which in an arbitrary numeric index.

5.5 Ordinary least squares and `lm()`

Since this section centers on the frequentist `lm()` function, I'm going to largely ignore it. A couple things, though. You'll note how the `brms` package uses the `lm()`-like design formula syntax. Although not as pedagogical as the more formal `rethinking` syntax, it has the advantage of cohering with the popular `lme4` syntax for multilevel models.

Also, on page 161 McElreath clarified that one cannot use the `I()` syntax with his `rethinking` package. Not so with `brms`. The `I()` syntax works just fine with `brms::brm()`. We've already made use of it in the "Polynomial regression" section of Chapter 4.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
```

```
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets  methods
## [8] base
##
## other attached packages:
## [1] GGally_1.4.0      tidybayes_1.0.4    bayesplot_1.6.0
## [4] fiftystater_1.0.1 ggrepel_0.8.0    forcats_0.3.0
## [7] stringr_1.4.0    dplyr_0.8.0.1   purrr_0.2.5
## [10] readr_1.1.1     tidyverse_1.2.1  Rcpp_1.0.1
## [13] tidyverse_1.2.1  brms_2.8.8       StanHeaders_2.18.0-1 ggplot2_3.1.1
## [16] rstan_2.18.2    StanHeaders_2.18.0-1 ggplot2_3.1.1
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2      ggridges_0.5.0
## [3] rsconnect_0.8.8       rprojroot_1.3-2
## [5] ggstance_0.3         markdown_0.8
## [7] base64enc_0.1-3      rstudioapi_0.7
## [9] svUnit_0.7-12        DT_0.4
## [11] fansi_0.4.0          mvtnorm_1.0-10
## [13] lubridate_1.7.4      xml2_1.2.0
## [15] bridgesampling_0.6-0 knitr_1.20
## [17] shinythemes_1.1.1    jsonlite_1.5
## [19] LaplacesDemon_16.1.1 broom_0.5.1
## [21] shiny_1.1.0          mapproj_1.2.6
## [23] compiler_3.5.1      httr_1.3.1
## [25] backports_1.1.4     assertthat_0.2.0
## [27] Matrix_1.2-14       lazyeval_0.2.2
## [29] cli_1.0.1           later_0.7.3
## [31] htmltools_0.3.6     prettyunits_1.0.2
## [33] tools_3.5.1          igraph_1.2.1
## [35] coda_0.19-2         gtable_0.3.0
## [37] glue_1.3.1.9000     reshape2_1.4.3
## [39] maps_3.3.0          cellranger_1.1.0
## [41] nlme_3.1-137        crosstalk_1.0.0
## [43] xfun_0.3            ps_1.2.1
## [45] rvest_0.3.2         mime_0.5
## [47] miniUI_0.1.1.1     gtools_3.8.1
## [49] MASS_7.3-50         zoo_1.8-2
## [51] scales_1.0.0        colourpicker_1.0
## [53] hms_0.4.2           promises_1.0.1
## [55] Brobdingnag_1.2-6   inline_0.3.15
## [57] RColorBrewer_1.1-2  shinystan_2.5.0
## [59] yaml_2.1.19         gridExtra_2.3
## [61] loo_2.1.0            reshape_0.8.7
## [63] stringi_1.4.3       dygraphs_1.1.1.5
## [65] pkgbuild_1.0.2       rlang_0.3.4
## [67] pkgconfig_2.0.2      matrixStats_0.54.0
## [69] HDInterval_0.2.0    evaluate_0.10.1
## [71] lattice_0.20-35     rstantools_1.5.1
## [73] htmlwidgets_1.2      labeling_0.3
## [75] processx_3.2.1     tidyselect_0.2.5
## [77] plyr_1.8.4          magrittr_1.5
## [79] bookdown_0.9         R6_2.3.0
## [81] generics_0.0.2       pillar_1.3.1
## [83] haven_1.1.2          withr_2.1.2
## [85] xts_0.10-2           abind_1.4-5
## [87] modelr_0.1.2         crayon_1.3.4
## [89] arrayhelpers_1.0-20160527 utf8_1.1.4
```

```
## [91] rmarkdown_1.10          grid_3.5.1
## [93] readxl_1.1.0            callr_3.1.0
## [95] threejs_0.3.1           digest_0.6.18
## [97] xtable_1.8-2             httpuv_1.4.4.2
## [99] stats4_3.5.1            munsell_0.5.0
## [101] shinyjs_1.0
```


Chapter 6

Overfitting, Regularization, and Information Criteria

In this chapter we contend with two contrasting kinds of statistical error:

- overfitting, “which leads to poor prediction by learning too *much* from the data”
- underfitting, “which leads to poor prediction by learning too *little* from the data” (p. 166, *emphasis added*)

6.1 The problem with parameters

The R^2 is a popular way to measure how well you can retrodict the data. It traditionally follows the form

$$R^2 = \frac{\text{var}(\text{outcome}) - \text{var}(\text{residuals})}{\text{var}(\text{outcome})} = 1 - \frac{\text{var}(\text{residuals})}{\text{var}(\text{outcome})}$$

By `var()`, of course, we meant variance (i.e., the `var()` function in R).

McElreath’s not a fan of the R^2 . But it’s important in my field, so instead of a summary at the end of the chapter, we will cover the Bayesian version of R^2 and how to use it in `brms`.

6.1.1 More parameters always improve fit.

We’ll start off by making the data with brain size and body size for seven `species`.

```
library(tidyverse)
(
  d <-
  tibble(species = c("afarensis", "africanus", "habilis", "boisei", "rudolfensis", "ergaster", "sapiens"),
         brain    = c(438, 452, 612, 521, 752, 871, 1350),
         mass     = c(37.0, 35.5, 34.5, 41.5, 55.5, 61.0, 53.5))
)

## # A tibble: 7 x 3
##   species      brain   mass
##   <chr>        <dbl>  <dbl>
## 1 afarensis     438    37
## 2 africanus     452    35.5
## 3 habilis       612    34.5
## 4 boisei        521    41.5
## 5 rudolfensis   752    55.5
## 6 ergaster       871    61
## 7 sapiens      1350   53.5
```

Let's get ready for Figure 6.2. The plots in this chapter will be characterized by `theme_classic() + theme(text = element_text(family = "Courier"))`. Our color palette will come from the [rcartocolor package](#), which provides color schemes designed by 'CARTO'.

```
# install.packages("rcartocolor", dependencies = T)
library(rcartocolor)
```

The specific palette we'll be using is "BurgYl." In addition to palettes, the rcartocolor package offers a few convenience functions which make it easier to use their palettes. The `carto_pal()` function will return the HEX numbers associated with a given palette's colors and the `display_carto_pal()` function will display the actual colors.

```
carto_pal(7, "BurgYl")
```

```
## [1] "#fbe6c5" "#f5ba98" "#ee8a82" "#dc7176" "#c8586c" "#9c3f5d" "#70284a"
```

```
display_carto_pal(7, "BurgYl")
```

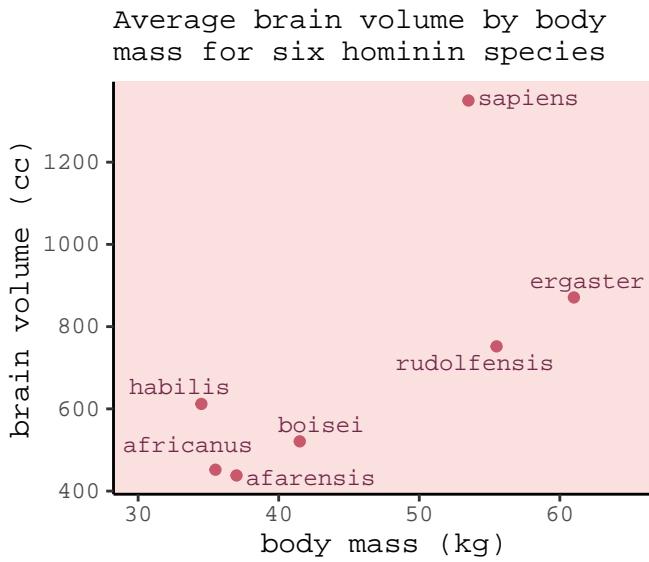
BurgYl: quantitative (7)



We'll be using a diluted version of the third color for the panel background (i.e., `theme(panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))`) and the darker purples for other plot elements. Here's the plot.

```
library(ggrepel)

d %>%
  ggplot(aes(x = mass, y = brain, label = species)) +
  geom_point(color = carto_pal(7, "BurgYl")[5]) +
  geom_text_repel(size = 3, color = carto_pal(7, "BurgYl")[7], family = "Courier", seed = 438) +
  coord_cartesian(xlim = 30:65) +
  labs(x = "body mass (kg)",
       y = "brain volume (cc)",
       subtitle = "Average brain volume by body\nmass for six hominin species") +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))
```



Let's fit the first six models in bulk. First we'll make a custom function, `fit_lm()`, into which we'll feed the desired names and formulas of our models. We'll make a tibble initially composed of those names (i.e., `model`) and formulas (i.e., `formula`). Via `purrr::map2()` within `mutate()`, we'll then fit the models and save the model objects within the tibble. The `broom package` provides an array of convenience functions to convert statistical analysis summaries into tidy data objects. We'll employ `broom::tidy()` and `broom::glance()` to extract information from the model fits.

```
library(broom)

fit_lm <- function(model, formula){
  model <- lm(data = d, formula = formula)
}

fits <-
  tibble(model  = str_c("b6.", 1:6),
         formula = c("brain ~ mass",
                     "brain ~ mass + I(mass^2)",
                     "brain ~ mass + I(mass^2) + I(mass^3)",
                     "brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4)",
                     "brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4) + I(mass^5)",
                     "brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4) + I(mass^5) + I(mass^6)") %>%
  mutate(fit    = map2(model, formula, fit_lm)) %>%
  mutate(tidy   = map(fit, tidy),
         glance = map(fit, glance))

# what did we just do?
print(fits)
```

```
## # A tibble: 6 x 5
##   model formula
##   <chr> <chr>
## 1 b6.1  brain ~ mass
## 2 b6.2  brain ~ mass + I(mass^2)
## 3 b6.3  brain ~ mass + I(mass^2) + I(mass^3)
## 4 b6.4  brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4)
## 5 b6.5  brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4) + I(mass^5)
## 6 b6.6  brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4) + I(mass^5) + I(mass^6)
```

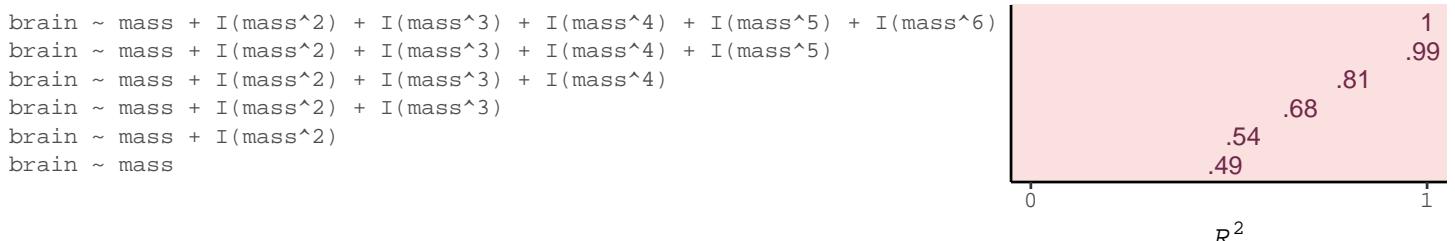
	fit	tidy	glance
1	<list>	<list>	<list>
2	<S3: 1~ <tibble [2 x~ <tibble [1 x~		
3	<S3: 1~ <tibble [3 x~ <tibble [1 x~		
4	<S3: 1~ <tibble [4 x~ <tibble [1 x~		
5	<S3: 1~ <tibble [5 x~ <tibble [1 x~		
6	<S3: 1~ <tibble [6 x~ <tibble [1 x~		

Our `fits` object is a `nested tibble`. To learn more about this bulk approach to fitting models, check out Hadley Wickham's talk [Managing many models with R](#). As you might learn in the talk, we can extract the R^2 from each model with `map_dbl("r.squared")`, which we'll then display in a plot.

```

fits <-
  fits %>%
    mutate(r2      = glance %>% map_dbl("r.squared")) %>%
    mutate(r2_text = round(r2, digits = 2) %>% as.character() %>% str_replace(., "0.", "."))
  fits %>%
    ggplot(aes(x = r2, y = formula, label = r2_text)) +
    geom_text(color = carto_pal(7, "BurgYl")[7], size = 3.5) +
    scale_x_continuous(expression(italic(R)^2), limits = 0:1, breaks = 0:1) +
    ylab(NULL) +
    theme_classic() +
    theme(axis.text.y = element_text(hjust = 0),
          axis.ticks.y = element_blank(),
          text        = element_text(family = "Courier"),
          panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))

```



If we wanted to look at the model coefficients, we could `unnest(tidy)` and wrangle a bit.

```

fits %>%
  unnest(tidy) %>%
  select(model, term:estimate) %>%
  mutate_if(is.double, round, digits = 1) %>%
  complete(term = distinct(., term), model) %>%
  spread(key = term, value = estimate) %>%
  select(model, -(Intercept)^, mass, everything())

```

```

## # A tibble: 6 x 8
##   model `-(Intercept)` mass `I(mass^2)` `I(mass^3)` `I(mass^4)` `I(mass^5)` `I(mass^6)`
##   <chr>       <dbl>   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 b6.1       -228.    20.7      NA        NA        NA        NA
## 2 b6.2      -2618.   127.     -1.1      NA        NA        NA
## 3 b6.3      21990.  -1474.    32.8     -0.2      NA        NA
## 4 b6.4      322887. -27946.   892.     -12.4     0.1       NA
## 5 b6.5     -1535342. 180049. -8325.    190.     -2.1       0
## 6 b6.6      10849891. -1473228. 82777.   -2463.    40.9     -0.4

```

For Figure 6.3, we'll make each plot individually and then glue them together with `gridExtra::grid.arrange()`. Since they all share a common structure, we'll start by specifying a base plot which we'll save as `p`.

```

p <-
  d %>%
    ggplot(aes(x = mass, y = brain)) +
    geom_point(color = carto_pal(7, "BurgYl")[7]) +
    scale_x_continuous("body mass (kg)", limits = c(33, 62), expand = c(0, 0)) +
    coord_cartesian(ylim = c(300, 1500)) +
    ylab("brain volume (cc)") +
    theme_classic() +
    theme(text = element_text(family = "Courier"),
          panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))

```

Now for each subplot, we'll tack the subplot-specific components onto `p`. The main action is in `stat_smooth()`. For each subplot, the first three lines in `stat_smooth()` are identical, with only the bottom `formula` line differing. Like McElreath did in the text, we also adjust the y-axis range for the last two plots.

```
# linear
p1 <-
  p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89, # note our rare use of 89% intervals
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ x) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = .49")))

# quadratic
p2 <-
  p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ poly(x, 2)) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = .54")))

# cubic
p3 <-
  p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ poly(x, 3)) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = .68")))

# fourth-order polynomial
p4 <-
  p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ poly(x, 4)) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = .81")))

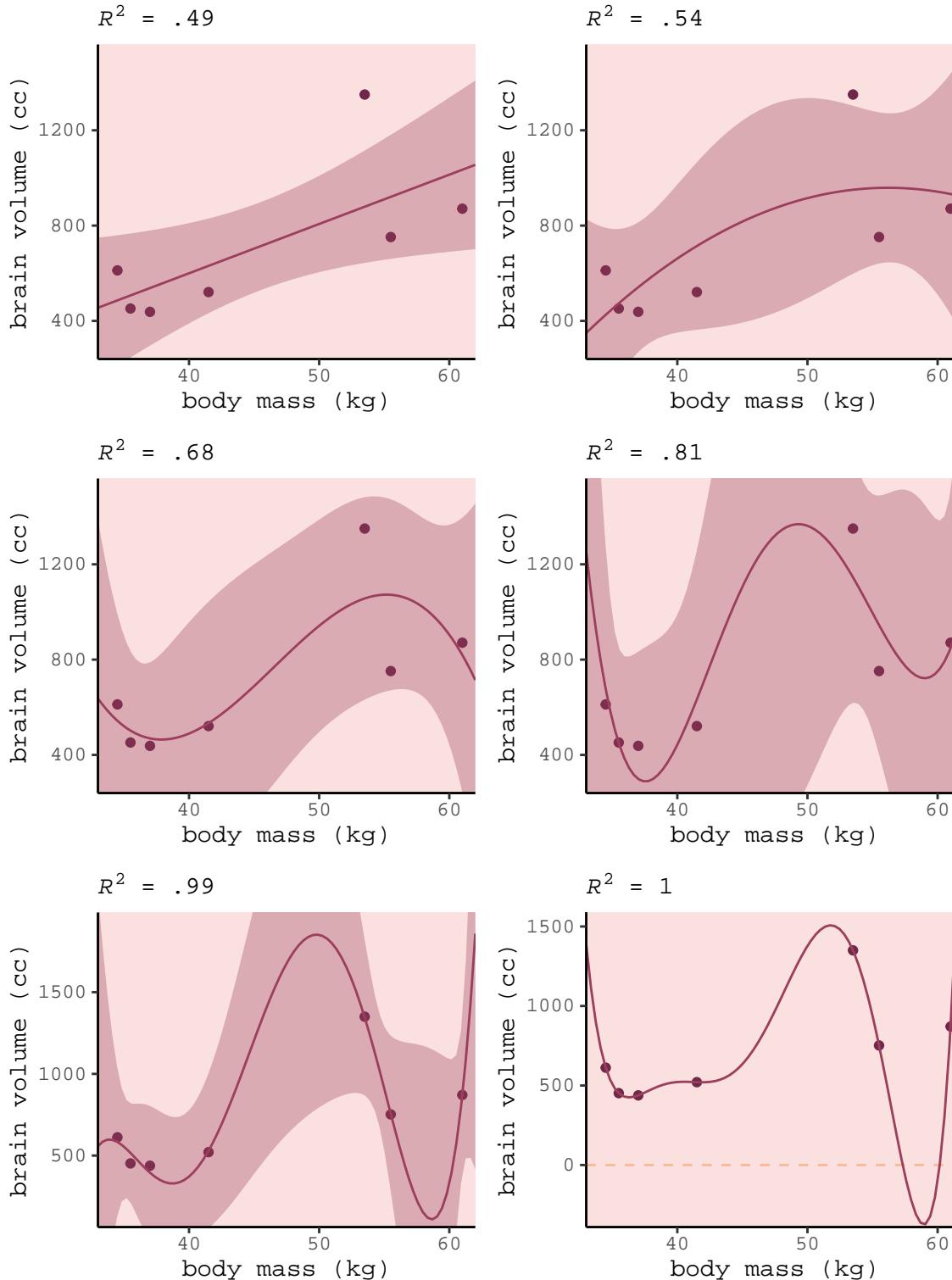
# fifth-order polynomial
p5 <-
  p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ poly(x, 5)) +
# we're adjusting the y-axis range for this plot (and the next)
coord_cartesian(ylim = c(150, 1900)) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = .99")))

# sixth-order polynomial
p6 <-
  p +
  # mark off 0 on the y-axis
  geom_hline(yintercept = 0, color = carto_pal(7, "BurgYl")[2], linetype = 2) +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ poly(x, 6)) +
```

```
coord_cartesian(ylim = c(-300, 1500)) +
ggttitle(NULL, subtitle = expression(paste(italic(R)^2, " = 1")))
```

Okay, now we're ready to combine the six subplots and produce our version of Figure 6.3.

```
library(gridExtra)
grid.arrange(p1, p2, p3, p4, p5, p6, ncol = 2)
```



6.1.2 Too few parameters hurts, too.

Fit the intercept only model, b6.7.

```
b6.7 <- lm(data = d,
            brain ~ 1)

summary(b6.7)

## 
## Call:
## lm(formula = brain ~ 1, data = d)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -275.71 -227.21 -101.71   97.79  636.29
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  713.7      121.8     5.86  0.00109 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 322.2 on 6 degrees of freedom
```

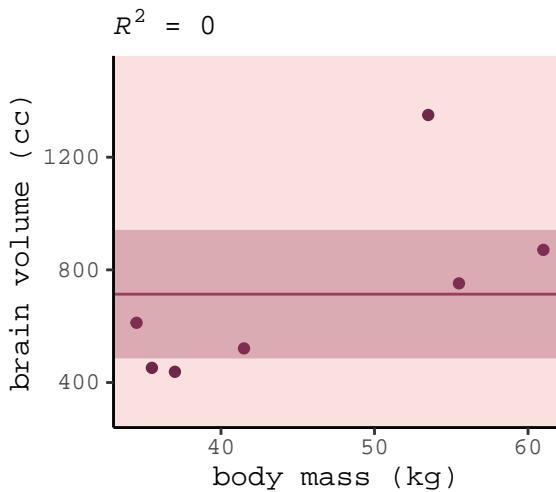
With the intercept-only model, we didn't even get an R^2 value in the `summary.broom::glance()` offers a quick way to get one.

```
glance(b6.7)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC deviance df.residual
##       <dbl>           <dbl>  <dbl>     <dbl>    <dbl> <int>  <dbl>  <dbl>  <dbl>    <dbl>        <int>
## 1       0            0    322.       NA      NA     1   -49.8  104.  104.  623061.          6
```

Zero. Our intercept-only b6.7 explained exactly zero variance in `brain`. All it did was tell us what the unconditional mean and variance (i.e., ‘Residual standard error’) were. I hope that makes sense. They were the only things in the model: $\text{brain}_i \sim \text{Normal}(\mu = \alpha, \sigma)$. To get the intercept-only model for Figure 6.4, we plug `formula = y ~ 1` into the `stat_smooth()` function.

```
p +
  stat_smooth(method = "lm", fullrange = TRUE, level = .89,
              color = carto_pal(7, "BurgYl")[6], fill = carto_pal(7, "BurgYl")[6],
              size = 1/2, alpha = 1/3,
              formula = y ~ 1) +
  ggtitle(NULL, subtitle = expression(paste(italic(R)^2, " = 0")))
```



6.1.2.1 Overthinking: Dropping rows.

You can `filter()` by `row_number()` to drop rows in a [tidyverse kind of way](#). For example, we can drop the second row of `d` like this.

```
d %>%
  filter(row_number() != 2)
```

```
## # A tibble: 6 x 3
##   species    brain  mass
##   <chr>      <dbl> <dbl>
## 1 afarensis    438   37
## 2 habilis      612   34.5
## 3 boisei        521   41.5
## 4 rudolfensis   752   55.5
## 5 ergaster       871   61
## 6 sapiens     1350   53.5
```

We can then extend that logic into a custom function, `make_lines()`, that will drop a row from `d`, fit the simple model `brain ~ mass`, and then use base R `predict()` to return the model-implied trajectory over new data values.

```
# because these lines are straight, we only need new data over two points of `mass`
nd <- tibble(mass = c(30, 70))

make_lines <- function(row){
  my_fit <-
  d %>%
    filter(row_number() != row) %>%
    lm(formula = brain ~ mass)

  predict(my_fit, nd) %>%
    as_tibble() %>%
    rename(brain = value) %>%
    bind_cols(nd)
}
```

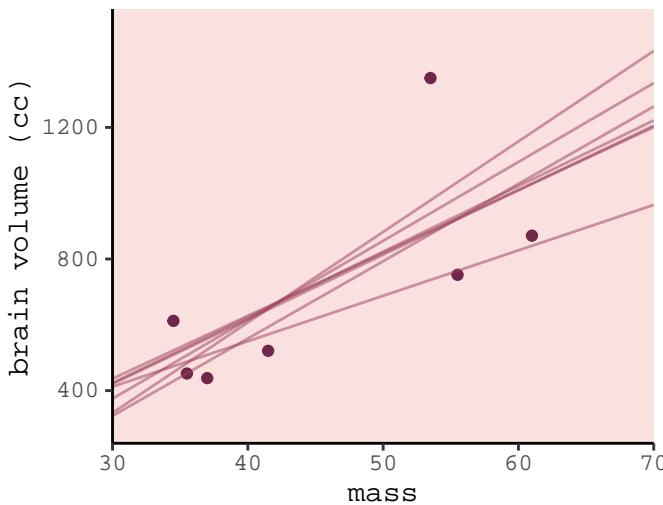
Here we'll make a tibble, `lines`, which will specify rows 1 through 7 in the `row` column. We'll then feed those `row` numbers into our custom `make_lines()` function, which will return the predicted values and their corresponding `mass` values, per model.

```
(  
  lines <-  
    tibble(row = 1:7) %>%  
    mutate(p = map(row, make_lines)) %>%  
    unnest(p)  
)
```

```
## # A tibble: 14 x 3  
##   row brain mass  
##   <int> <dbl> <dbl>  
## 1     1  436.   30  
## 2     1 1201.   70  
## 3     2  421.   30  
## 4     2 1205.   70  
## 5     3  323.   30  
## 6     3 1264.   70  
## 7     4  423.   30  
## 8     4 1221.   70  
## 9     5  376.   30  
## 10    5 1335.   70  
## 11    6  332.   30  
## 12    6 1433.   70  
## 13    7  412.   30  
## 14    7  964.   70
```

Now we're ready to plot the left panel of Figure 6.5.

```
p +  
  scale_x_continuous(expand = c(0, 0)) +  
  geom_line(data = lines,  
            aes(x = mass, y = brain, group = row),  
            color = carto_pal(7, "BurgYl")[6], alpha = 1/2, size = 1/2)
```



To make the right panel for Figure 6.5, we'll need to increase the number of `mass` points in our `nd` data and redefine the `make_lines()` function to fit the sixth-order-polynomial model.

```
# because these lines will be very curvy, we'll need new data over many points of `mass`  
nd <- tibble(mass = seq(from = 30, to = 65, length.out = 200))  
  
# redefine the function  
make_lines <- function(row){  
  my_fit <-
```

```

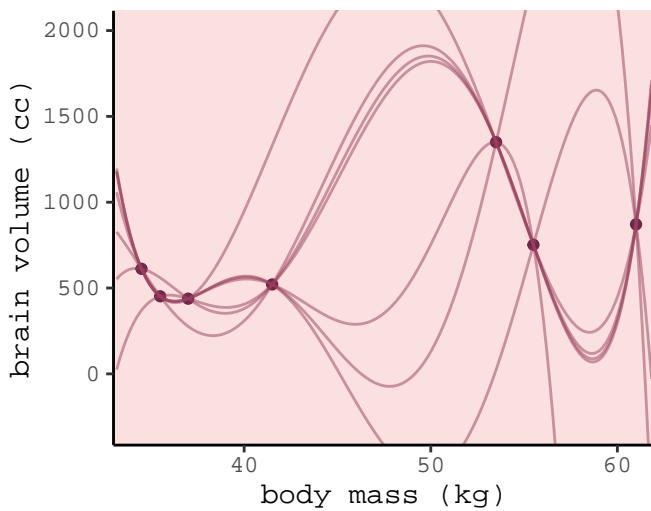
d %>%
  filter(row_number() != row) %>%
  lm(formula = brain ~ mass + I(mass^2) + I(mass^3) + I(mass^4) + I(mass^5) + I(mass^6))

predict(my_fit, nd) %>%
  as_tibble() %>%
  rename(brain = value) %>%
  bind_cols(nd)
}

# make our new tibble
lines <-
  tibble(row = 1:7) %>%
  mutate(p = map(row, make_lines)) %>%
  unnest(p)

# plot!
p +
  geom_line(data = lines,
            aes(group = row),
            color = carto_pal(7, "BurgYL")[6], alpha = 1/2, size = 1/2) +
  coord_cartesian(ylim = -300:2000)

```



6.2 Information theory and model performance

Whether you end up using regularization or information criteria or both, the first thing you must do is pick a criterion of model performance. What do you want the model to do well at? We'll call this criterion the *target*, and in this section you'll see how information theory provides a common and useful target, the out-of-sample *deviance*. (p. 174, *emphasis* in the original)

6.2.1 Firing the weatherperson.

If you let rain = 1 and sun = 0, here's a way to make a plot of the first table of page 175, the weatherperson's predictions.

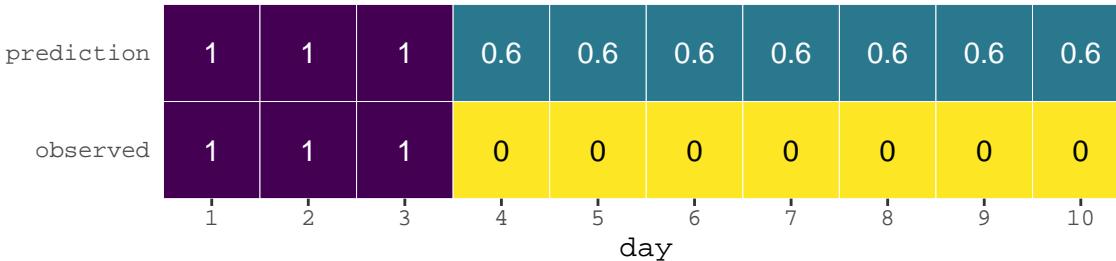
```

weatherperson <-
  tibble(day      = 1:10,
        prediction = rep(c(1, 0.6), times = c(3, 7)),
        observed   = rep(c(1, 0), times = c(3, 7)))

weatherperson %>%

```

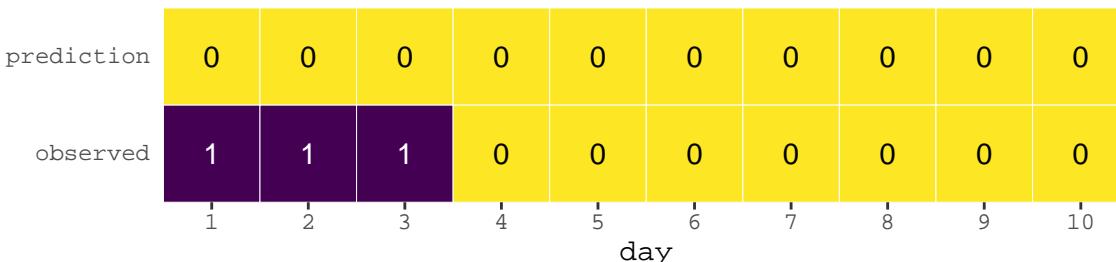
```
gather(key, value, -day) %>%
  ggplot(aes(x = day, y = key, fill = value)) +
  geom_tile(color = "white") +
  geom_text(aes(label = value, color = value == 0)) +
  scale_x_continuous(breaks = 1:10, expand = c(0, 0)) +
  scale_y_discrete(NULL, expand = c(0, 0)) +
  scale_fill_viridis_c(direction = -1) +
  scale_color_manual(values = c("white", "black")) +
  theme(legend.position = "none",
        axis.ticks.y = element_blank(),
        text = element_text(family = "Courier"))
```



Here's how the newcomer fared:

```
newcomer <-
  tibble(day      = 1:10,
         prediction = 0,
         observed   = rep(c(1, 0), times = c(3, 7)))

newcomer %>%
  gather(key, value, -day) %>%
  ggplot(aes(x = day, y = key, fill = value)) +
  geom_tile(color = "white") +
  geom_text(aes(label = value, color = value == 0)) +
  scale_x_continuous(breaks = 1:10, expand = c(0, 0)) +
  scale_y_discrete(NULL, expand = c(0, 0)) +
  scale_fill_viridis_c(direction = -1) +
  scale_color_manual(values = c("white", "black")) +
  theme(legend.position = "none",
        axis.ticks.y = element_blank(),
        text = element_text(family = "Courier"))
```



If we do the math entailed in the tibbles, we'll see why the newcomer could boast "I'm the best person for the job" (p. 175).

```
weatherperson %>%
  bind_rows(newcomer) %>%
  mutate(person = rep(c("weatherperson", "newcomer"), each = n()/2),
         hit     = ifelse(prediction == observed, 1, 1 - prediction - observed)) %>%
  group_by(person) %>%
  summarise(hit_rate = mean(hit))
```

```
## # A tibble: 2 x 2
##   person      hit_rate
##   <chr>       <dbl>
## 1 newcomer     0.7
## 2 weatherperson 0.58
```

6.2.1.1 Costs and benefits.

Our new points variable doesn't fit into the nice color-based `geom_tile()` plots from above. But we can still do the math.

```
weatherperson %>%
  bind_rows(newcomer) %>%
  mutate(person = rep(c("weatherperson", "newcomer"), each = n() / 2),
         points = ifelse(observed == 1 & prediction != 1, -5,
                         ifelse(observed == 1 & prediction == 1, -1,
                               -1 * prediction))) %>%
  group_by(person) %>%
  summarise(happiness = sum(points))
```

```
## # A tibble: 2 x 2
##   person      happiness
##   <chr>       <dbl>
## 1 newcomer     -15
## 2 weatherperson -7.2
```

6.2.1.2 Measuring accuracy.

```
weatherperson %>%
  bind_rows(newcomer) %>%
  mutate(person = rep(c("weatherperson", "newcomer"), each = n() / 2),
         hit = ifelse(prediction == observed, 1, 1 - prediction - observed)) %>%
  group_by(person, hit) %>%
  count() %>%
  ungroup() %>%
  mutate(power = hit ^ n,
         term = rep(letters[1:2], times = 2)) %>%
  select(person, term, power) %>%
  spread(key = term, value = power) %>%
  mutate(probability_correct_sequence = a * b)
```

```
## # A tibble: 2 x 4
##   person      a      b probability_correct_sequence
##   <chr>     <dbl> <dbl>                  <dbl>
## 1 newcomer     0      1                      0
## 2 weatherperson 0.00164    1                  0.00164
```

6.2.2 Information and uncertainty.

The formula for information entropy is:

$$H(p) = -E \log(p_i) = -\sum_{i=1}^n p_i \log(p_i)$$

McElreath put it in words as “the uncertainty contained in a probability distribution is the average log-probability of the event” (p. 178). We’ll compute the information entropy for weather at the first unnamed location, which we’ll call McElreath’s house, and Abu Dhabi at once.

```
tibble(place = c("McElreath's house", "Abu Dhabi"),
      p_rain = c(.3, .01)) %>%
  mutate(p_shine = 1 - p_rain) %>%
  group_by(place) %>%
  mutate(H_p = (p_rain * log(p_rain) + p_shine * log(p_shine)) %>% mean() * -1)
```

```
## # A tibble: 2 x 4
## # Groups: place [2]
##   place          p_rain p_shine    H_p
##   <chr>        <dbl>   <dbl>    <dbl>
## 1 McElreath's house     0.3     0.7  0.611
## 2 Abu Dhabi           0.01    0.99 0.0560
```

The uncertainty is less in Abu Dhabi because it rarely rains, there. If you have sun, rain and snow, the entropy for weather is:

```
p <- c(.7, .15, .15)
-sum(p * log(p))
```

```
## [1] 0.8188085
```

6.2.3 From entropy to accuracy.

The formula for the Kullback-Leibler divergence (i.e., K-L divergence) is

$$D_{\text{KL}}(p, q) = \sum_i p_i (\log(p_i) - \log(q_i)) = \sum_i p_i \log\left(\frac{p_i}{q_i}\right)$$

which, in plainer language, is what McElreath described as “the average difference in log probability between the target (p) and model (q)” (p. 179).

In McElreath’s example

- $p_1 = .3$
- $p_2 = .7$
- $q_1 = .25$
- $q_2 = .75$

With those values, we can compute $D_{\text{KL}}(p, q)$ within a tibble like so:

```
tibble(p_1 = .3,
       p_2 = .7,
       q_1 = .25,
       q_2 = .75) %>%
  mutate(d_kl = (p_1 * log(p_1 / q_1)) + (p_2 * log(p_2 / q_2)))
```



```
## # A tibble: 1 x 5
##   p_1   p_2   q_1   q_2   d_kl
##   <dbl> <dbl> <dbl> <dbl>   <dbl>
## 1 0.3   0.7   0.25  0.75  0.00640
```

Our systems in this section are binary (e.g., $q = \{q_1, q_2\}$). Thus if you know $q_1 = .3$ you know of a necessity $q_2 = 1 - q_1$. Therefore we can code the tibble for the next example of when $p = q$ like this:

```
tibble(p_1 = .3) %>%
  mutate(p_2 = 1 - p_1,
        q_1 = p_1) %>%
  mutate(q_2 = 1 - q_1) %>%
  mutate(d_kl = (p_1 * log(p_1 / q_1)) + (p_2 * log(p_2 / q_2)))
```

```
## # A tibble: 1 x 5
##       p_1     p_2     q_1     q_2   d_kl
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1     0.3     0.7     0.3     0.7     0
```

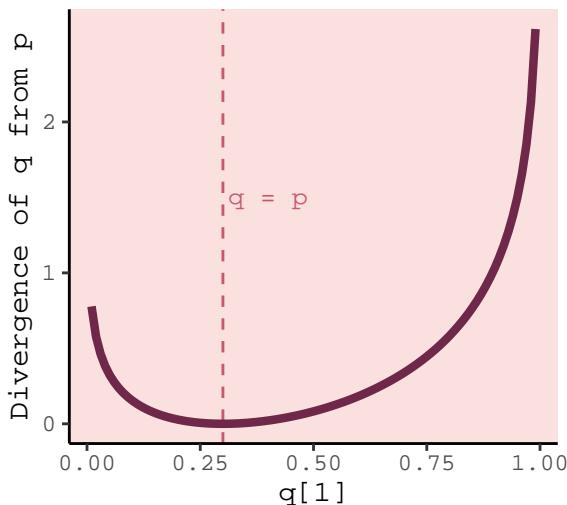
Building off of that, you can make the data required for Figure 6.6 like this.

```
t <-  
  tibble(p_1 = .3,  
         p_2 = .7,  
         q_1 = seq(from = .01, to = .99, by = .01)) %>%  
  mutate(q_2 = 1 - q_1) %>%  
  mutate(d_kl = (p_1 * log(p_1 / q_1)) + (p_2 * log(p_2 / q_2)))  
  
head(t)
```

```
## # A tibble: 6 x 5
##       p_1     p_2     q_1     q_2   d_kl
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1     0.3     0.7     0.01    0.99  0.778
## 2     0.3     0.7     0.02    0.98  0.577
## 3     0.3     0.7     0.03    0.97  0.462
## 4     0.3     0.7     0.04    0.96  0.383
## 5     0.3     0.7     0.05    0.95  0.324
## 6     0.3     0.7     0.06    0.94  0.276
```

Now we have the data, plotting Figure 6.6 is a just `geom_line()` with stylistic flourishes.

```
t %>%
  ggplot(aes(x = q_1, y = d_kl)) +
  geom_vline(xintercept = .3, color = carto_pal(7, "BurgYl")[5], linetype = 2) +
  geom_line(color = carto_pal(7, "BurgYl")[7], size = 1.5) +
  annotate(geom = "text", x = .4, y = 1.5, label = "q = p",
           color = carto_pal(7, "BurgYl")[5], family = "Courier", size = 3.5) +
  labs(x = "q[1]",  
       y = "Divergence of q from p") +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))
```



6.2.3.1 Rethinking: Divergence depends upon direction.

Here we see $H(p, q) \neq H(q, p)$. That is, direction matters.

```
tibble(direction = c("Earth to Mars", "Mars to Earth"),
       p_1      = c(.01, .7),
       q_1      = c(.7, .01)) %>%
  mutate(p_2 = 1 - p_1,
        q_2 = 1 - q_1) %>%
  mutate(d_kl = (p_1 * log(p_1 / q_1)) + (p_2 * log(p_2 / q_2)))
```

```
## # A tibble: 2 x 6
##   direction      p_1     q_1     p_2     q_2   d_kl
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>   <dbl>
## 1 Earth to Mars 0.01    0.7     0.99    0.3    1.14
## 2 Mars to Earth 0.7     0.01    0.3     0.99   2.62
```

The D_{KL} was double when applying Martian estimates to Terran estimates.

6.2.4 From divergence to deviance.

The point of all the preceding material about information theory and divergence is to establish both:

1. How to measure the distance of a model from our target. Information theory gives us the distance measure we need, the K-L divergence.
2. How to estimate the divergence. Having identified the right measure of distance, we now need a way to estimate it in real statistical modeling tasks. (p. 181)

Now we'll start working on item #2.

We define deviance as:

$$D(q) = -2 \sum_i \log(p_i)$$

In the formula, i indexes each case and q_i is the likelihood for each case. Here's the deviance from model b6.1.

```
lm(data = d,
   brain ~ mass) %>%
  logLik() * -2

## 'log Lik.' 94.92499 (df=3)
```

6.2.4.1 Overthinking: Computing deviance.

To follow along with the text, we'll need to standardize `mass` before we compute deviance.

```
d <-
  d %>%
  mutate(mass_s = (mass - mean(mass)) / sd(mass))
```

Open brms.

```
library(brms)
```

Now we'll specify the initial values and fit the model.

```
# Here we specify our starting values
inits <- list(Intercept = mean(d$brain),
              mass_s      = 0,
              sigma       = sd(d$brain))

inits_list <- list(inits, inits, inits, inits)

# The model
b6.8 <-
  brm(data = d, family = gaussian,
       brain ~ 1 + mass_s,
       prior = c(prior(normal(0, 1000), class = Intercept),
                  prior(normal(0, 1000), class = b),
                  prior(cauchy(0, 10), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       inits = inits_list, # here we insert our start values
       seed = 6)
```

```
print(b6.8)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: brain ~ 1 + mass_s
##   Data: d (Number of observations: 7)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept    709.49     105.07    502.06    922.93        2804 1.00
## mass_s       220.64     109.71     -0.21    439.82        2893 1.00
##
## Family Specific Parameters:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma     263.87      94.86    148.87    495.15        1746 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Details about `inits`: You don't have to specify your `inits` lists outside of the `brm()` function the way we did, here. This is just how I currently prefer. When you specify start values for the parameters in your Stan models, you need to do so with a list of lists. You need as many lists as HMC chains—four in this example. And then you put your—in this case—four lists inside a list. Lists within lists. Also, we were lazy and specified the same start values across all our chains. You can mix them up across chains if you want.

Anyway, the `brms` function `log_lik()` returns a matrix. Each occasion gets a column and each HMC chain iteration gets a row. To make it easier to understand the output, we'll name the columns by `species` using the `.name_repair` argument within the `as_tibble()` function.

```
ll <-  
  b6.8 %>%  
  log_lik() %>%  
  as_tibble(.name_repair = ~ d$species)  
  
ll %>%  
  glimpse()  
  
## Observations: 4,000  
## Variables: 7  
## $ afarensis    <dbl> -6.533346, -7.364236, -6.354499, -6.591231, -6.198465, -6.605723, -6.428844...  
## $ africanus   <dbl> -6.529212, -7.318601, -6.380887, -6.501201, -6.204924, -6.630343, -6.489288...  
## $ habilis     <dbl> -6.684437, -7.078882, -6.924421, -6.465999, -6.729976, -6.965095, -7.075861...  
## $ boisei       <dbl> -6.543591, -7.261918, -6.397131, -6.604107, -6.196515, -6.603495, -6.416063...  
## $ rudolfensis <dbl> -6.947669, -7.070620, -7.095121, -6.729509, -6.196600, -6.611958, -6.591881...  
## $ ergaster     <dbl> -7.387421, -7.009474, -7.415504, -6.694015, -6.203039, -6.609322, -6.669332...  
## $ sapiens     <dbl> -11.396992, -7.490887, -7.742427, -8.074139, -11.303673, -8.671387, -8.5760...
```

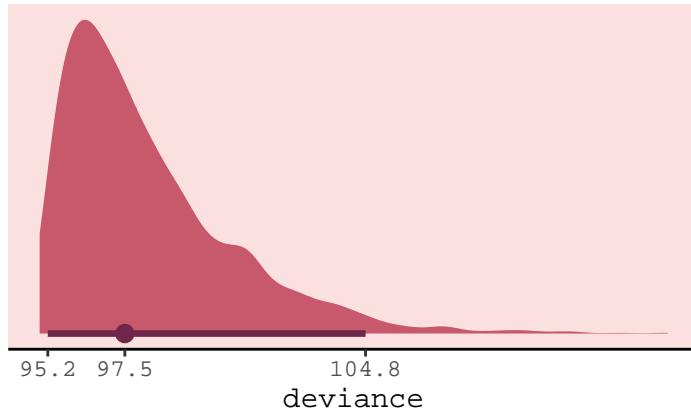
Deviance is the sum of the occasion-level LLs multiplied by -2. Why by -2? “The -2 in front doesn’t do anything important. It’s there for historical reasons” (p. 182). If you follow footnote 93 at the end of that sentence in the text, you’ll learn “under somewhat general conditions, for many common model types, a difference between two deviances has a chi-squared distribution. The factor of 2 is there to scale it that way” (p. 451).

```
ll <-  
  ll %>%  
  mutate(sums      = rowSums(.),  
         deviance = -2 * sums)
```

Because we used HMC, deviance is a distribution rather than a single number.

```
library(tidybayes)  
  
ll %>%  
  ggplot(aes(x = deviance, y = 0)) +  
  geom_halfeyeh(fill = carto_pal(7, "BurgY1")[5], color = carto_pal(7, "BurgY1")[7],  
                point_interval = median_qi, .width = .95) +  
  scale_x_continuous(breaks = quantile(ll$deviance, c(.025, .5, .975)),  
                     labels = quantile(ll$deviance, c(.025, .5, .975)) %>% round(1)) +  
  scale_y_continuous(NULL, breaks = NULL) +  
  labs(title = "The deviance distribution") +  
  theme_classic() +  
  theme(text = element_text(family = "Courier"),  
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgY1")[3], 1/4)))
```

The deviance distribution



But notice our deviance distribution was centered right around the sole value McElreath reported in the text.

6.2.5 From deviance to out-of-sample.

Deviance is a principled way to measure distance from the target. But deviance as computed in the previous section has the same flaw as R^2 : It always improves as the model gets more complex, at least for the types of models we have considered so far. Just like R^2 , deviance in-sample is a measure of retrodictive accuracy, not predictive accuracy.

In the next subsection, we'll see this in a simulation.

6.2.5.1 Overthinking: Simulated training and testing.

I find the `rethinking::sim.train.test()` function opaque. If you're curious, you can find McElreath's code [here](#). Let's simulate and see what happens.

```
library(rethinking)

n      <- 20
kseq   <- 1:5
# I've reduced this number by one order of magnitude to reduce computation time
n_sim  <- 1e3
n_cores <- 4

# here's our dev object based on `N <- 20`
dev_20 <-
  sapply(kseq, function(k) {
    print(k);
    r <- mcreplicate(n_sim, sim.train.test(N = n, k = k),
                      mc.cores = n_cores);
    c(mean(r[1,]), mean(r[2,]), sd(r[1,]), sd(r[2,]))
  })

# here's our dev object based on N <- 100
n      <- 100
dev_100 <-
  sapply(kseq, function(k) {
    print(k);
    r <- mcreplicate(n_sim, sim.train.test(N = n, k = k),
                      mc.cores = n_cores);
    c(mean(r[1,]), mean(r[2,]), sd(r[1,]), sd(r[2,]))
  })
```

If you didn't quite catch it, the simulation yields `dev_20` and `dev_100`. We'll want to convert them to tibbles, bind them together, and wrangle extensively before we're ready to plot.

```
dev_tibble <-
  dev_20 %>%
  as_tibble() %>%
  bind_rows(
    dev_100 %>%
    as_tibble()
  ) %>%
  mutate(n      = rep(c("n = 20", "n = 100"), each = 4),
        statistic = rep(c("mean", "sd"), each = 2) %>% rep(., times = 2),
        sample    = rep(c("in", "out"), times = 2) %>% rep(., times = 2)) %>%
  gather(n_par, value, -n, -statistic, -sample) %>%
  spread(key = statistic, value = value) %>%
  mutate(n      = factor(n, levels = c("n = 20", "n = 100")),
        n_par = str_remove(n_par, "V") %>% as.double() %>%
  mutate(n_par = ifelse(sample == "in", n_par - .075, n_par + .075))

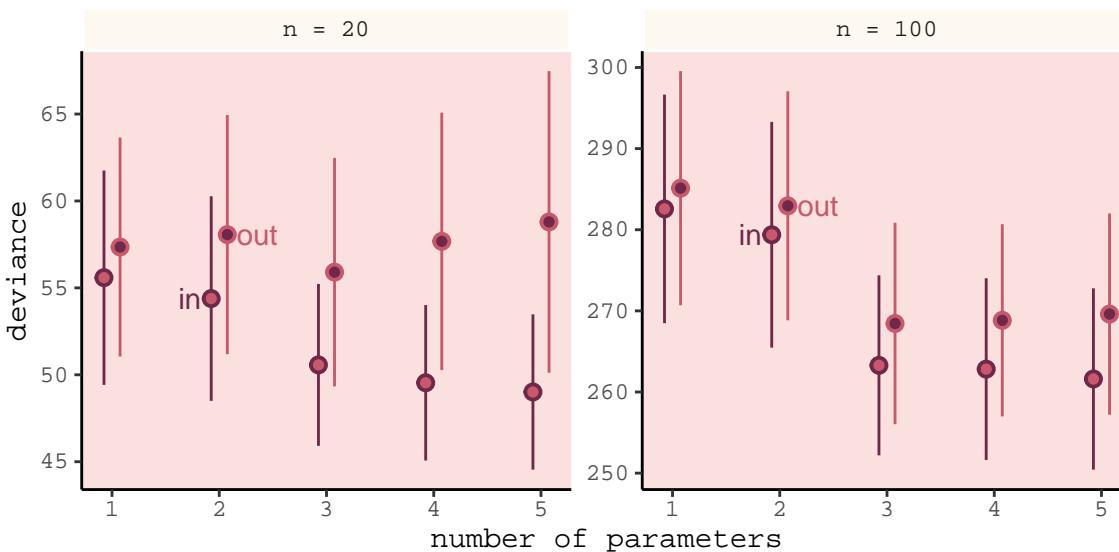
head(dev_tibble)

## # A tibble: 6 x 5
##   n     sample n_par  mean    sd
##   <fct>  <chr>  <dbl> <dbl> <dbl>
## 1 n = 100 in     0.925  283.  14.1
## 2 n = 100 in     1.92   279.  13.9
## 3 n = 100 in     2.92   263.  11.1
## 4 n = 100 in     3.92   263.  11.2
## 5 n = 100 in     4.92   262.  11.2
## 6 n = 100 out    1.08   285.  14.4
```

Now we're ready to make Figure 6.7.

```
# this intermediary tibble will make `geom_text()` easier
dev_text <-
  dev_tibble %>%
  filter(n_par > 1.5,
         n_par < 2.5) %>%
  mutate(n_par = ifelse(sample == "in", n_par - .2, n_par + .28))

# the plot
dev_tibble %>%
  ggplot(aes(x      = n_par, y = mean,
             ymin   = mean - sd, ymax = mean + sd,
             group  = sample,
             color  = sample,
             fill   = sample)) +
  geom_pointrange(shape = 21) +
  geom_text(data = dev_text,
            aes(label = sample)) +
  scale_color_manual(values = c(carto_pal(7, "BurgYl")[7], carto_pal(7, "BurgYl")[5])) +
  scale_fill_manual(values  = c(carto_pal(7, "BurgYl")[5], carto_pal(7, "BurgYl")[7])) +
  labs(x = "number of parameters",
       y = "deviance") +
  theme_classic() +
  theme(text          = element_text(family = "Courier"),
        legend.position = "none",
        strip.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[1], 1/4), color = "white"),
        panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[3], 1/4))) +
  facet_wrap(~n, scale = "free_y")
```



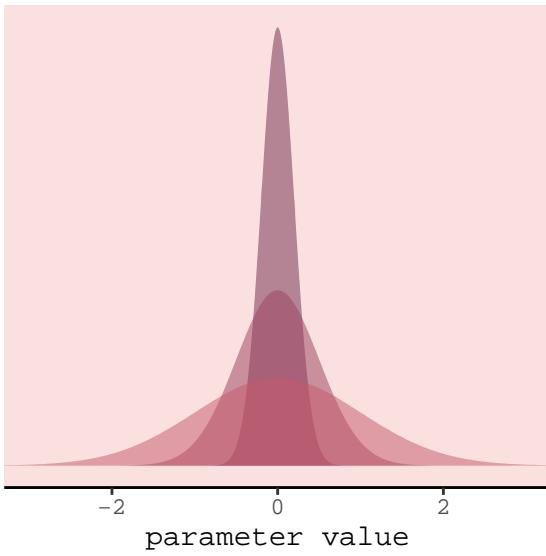
Even with a substantially smaller N , our simulation results matched up well with those in the text.

6.3 Regularization

The root of overfitting is a model's tendency to get overexcited by the training sample... One way to prevent a model from getting too excited by the training sample is to give it a skeptical prior. By "skeptical," I mean a prior that slows the rate of learning from the sample. (p. 186)

In case you were curious, here's how you might do Figure 6.8 with ggplot2. All the action is in the `geom_ribbon()` portions.

```
tibble(x = seq(from = - 3.5,
               to    = 3.5,
               by   = .01)) %>%
  ggplot(aes(x = x, ymin = 0)) +
  geom_ribbon(aes(ymax = dnorm(x, mean = 0, sd = 0.2)),
              fill = carto_pal(7, "BurgY1")[7], alpha = 1/2) +
  geom_ribbon(aes(ymax = dnorm(x, mean = 0, sd = 0.5)),
              fill = carto_pal(7, "BurgY1")[6], alpha = 1/2) +
  geom_ribbon(aes(ymax = dnorm(x, mean = 0, sd = 1)),
              fill = carto_pal(7, "BurgY1")[5], alpha = 1/2) +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab("parameter value") +
  coord_cartesian(xlim = c(-3, 3)) +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgY1")[3], 1/4)))
```



In our version of the plot, darker purple = more regularizing.

But to prepare for Figure 6.9, let's simulate. This time we'll wrap the basic simulation code we used before into a function we'll call `make_sim()`. Our `make_sim()` function has two parameters, `N` and `b_sigma`, both of which come from McElreath's simulation code. So you'll note that instead of hard coding the values for `N` and `b_sigma` within the simulation, we're leaving them adjustable (i.e., `sim.train.test(N = n, k = k, b_sigma = b_sigma)`). Also notice that instead of saving the simulation results as objects, like before, we're just converting them to tibbles with the `as_tibble()` function at the bottom. Our goal is to use `make_sim()` within a `purrr::map2()` statement. The result will be a nested tibble into which we've saved the results of 6 simulations based off of two sample sizes (i.e., `n = c(20, 100)`) and three values of σ for our Gaussian β prior (i.e., `b_sigma = c(1, .5, .2)`).

```
library(rethinking)

# I've reduced this number by one order of magnitude to reduce computation time
n_sim <- 1e3

make_sim <- function(n, b_sigma){
  sapply(kseq, function(k) {
    print(k);
    # this is an augmented line of code
    r <- mcreplicate(n_sim, sim.train.test(N = n, k = k, b_sigma = b_sigma),
                     mc.cores = n_cores);
    c(mean(r[1,]), mean(r[2,]), sd(r[1,]), sd(r[2,])) }) %>%
    # this is a new line of code
    as_tibble()
}

s <-
  tibble(n      = rep(c(20, 100), each = 3),
         b_sigma = rep(c(1, .5, .2), times = 2)) %>%
  mutate(sim    = map2(n, b_sigma, make_sim)) %>%
  unnest()
```

We'll follow the same principles for wrangling these data as we did those from the previous simulation, `dev_tibble`. And after wrangling, we'll feed the data directly into the code for our version of Figure 6.9.

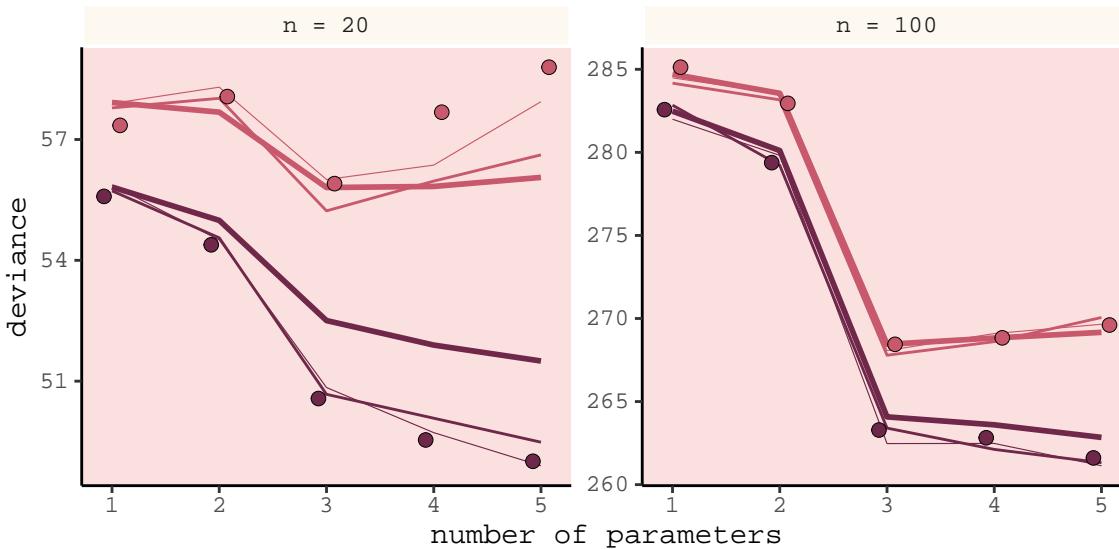
```
# wrangle the simulation data
s %>%
  mutate(statistic = rep(c("mean", "sd"), each = 2) %>% rep(., times = 3 * 2),
         sample   = rep(c("in", "out"), times = 2) %>% rep(., times = 3 * 2)) %>%
  gather(n_par, value, -n, -b_sigma, -statistic, -sample) %>%
```

```

spread(key = statistic, value = value) %>%
mutate(n      = str_c("n = ", n) %>% factor(. , levels = c("n = 20", "n = 100")),
n_par = str_remove(n_par, "V") %>% as.double() %>%

# now plot
ggplot(aes(x = n_par, y = mean,
            group = interaction(sample, b_sigma))) +
  geom_line(aes(color = sample, size = b_sigma %>% as.character())) +
# this function contains the data from the previous simulation
  geom_point(data = dev_tibble,
             aes(group = sample, fill = sample),
             color = "black", shape = 21, size = 2.5, stroke = .1) +
  scale_fill_manual(values = c(carto_pal(7, "BurgYl")[7], carto_pal(7, "BurgYl")[5])) +
  scale_color_manual(values = c(carto_pal(7, "BurgYl")[7], carto_pal(7, "BurgYl")[5])) +
  scale_size_manual(values = c(1, .5, .2)) +
  labs(x = "number of parameters",
       y = "deviance") +
  theme_classic() +
  theme(text          = element_text(family = "Courier"),
        legend.position = "none",
        strip.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[1], 1/4), color = "white"),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4))) +
  facet_wrap(~n, scale = "free_y")

```



Our results don't perfectly align with those in the text. I suspect this is because we used `1e3` iterations, rather than the `1e4` of the text. If you'd like to wait all night long for the simulation to yield more stable results, be my guest.

Regularizing priors are great, because they reduce overfitting. But if they are too skeptical, they prevent the model from learning from the data. So to use them effectively, you need some way to tune them. Tuning them isn't always easy. (p. 187)

For more on this how to choose your priors, consider Gelman, Simpson, and Betancourt's [The prior can generally only be understood in the context of the likelihood](#), a paper that will probably make more sense after Chapter 8. And if you're feeling feisty, also check out Simpson's related blog post ([It's never a Total Eclipse of the Prior](#)).

6.3.0.1 Rethinking: Ridge regression.

Within the brms framework, you can do something like this with the horseshoe prior via the `horseshoe()` function. You can learn all about it from the `horseshoe` section of the [brms reference manual \(version 2.8.0\)](#). Here's an extract from the section:

The horseshoe prior is a special shrinkage prior initially proposed by Carvalho et al. (2009). It is symmetric around zero with fat tails and an infinitely large spike at zero. This makes it ideal for sparse models that have many regression coefficients, although only a minority of them is non-zero. The horseshoe prior can be applied on all population-level effects at once (excluding the intercept) by using `set_prior("horseshoe(1)")`. (p. 70)

And to dive even deeper into the horseshoe prior, check out Michael Betancourt's tutorial, [Bayes Sparse Regression](#).

6.4 Information criteria

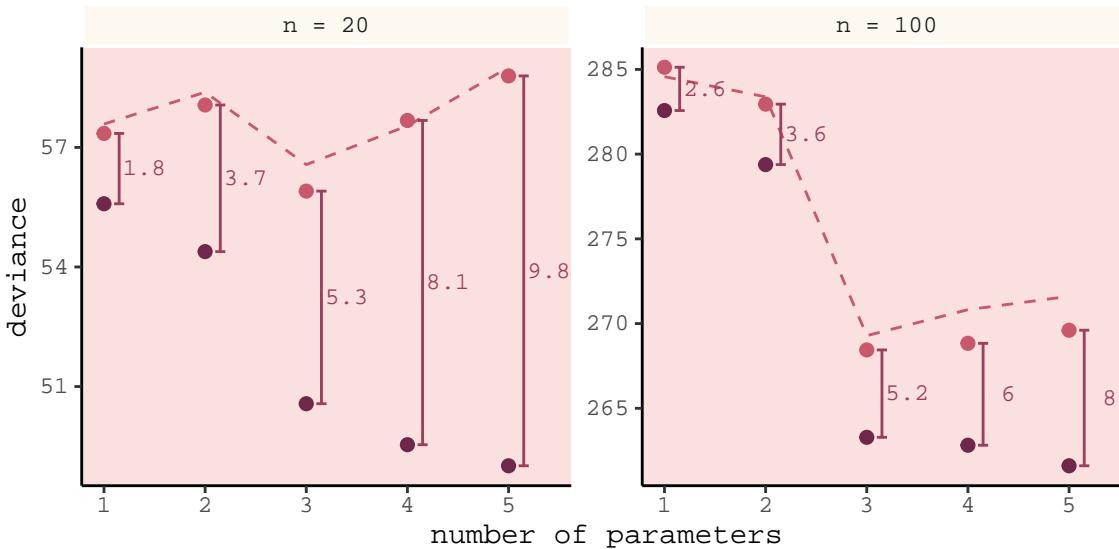
The data from our initial simulation isn't formatted well to plot Figure 6.10. We'll have to wrangle a little.

```
(  
  dev_tibble <-  
  dev_tibble %>%  
  select(-sd) %>%  
  mutate(n_par = ifelse(sample == "in", n_par + .075, n_par - .075)) %>%  
  spread(key = sample, value = mean) %>%  
  mutate(height = (out - `in`) %% round(digits = 1) %>% as.character(),  
         dash = `in` + 2 * n_par)  
)  
  
## # A tibble: 10 x 6  
##   n      n_par `in`    out height  dash  
##   <dbl>    <dbl> <dbl>  <dbl> <chr>  <dbl>  
## 1 20      1     55.6  57.4  1.8    57.6  
## 2 20      2     54.4  58.1  3.7    58.4  
## 3 20      3     50.6  55.9  5.3    56.6  
## 4 20      4     49.5  57.7  8.1    57.5  
## 5 20      5     49.0  58.8  9.8    59.0  
## 6 100     1     283.   285.   2.6    285.  
## 7 100     2     279.   283.   3.6    283.  
## 8 100     3     263.   268.   5.2    269.  
## 9 100     4     263.   269.   6      271.  
## 10 100    5     262.   270.   8      272.
```

Now we're ready to plot.

```
dev_tibble %>%  
  ggplot(aes(x = n_par)) +  
  geom_line(aes(y = dash),  
            linetype = 2, color = carto_pal(7, "BurgYl")[5]) +  
  geom_point(aes(y = `in`),  
             color = carto_pal(7, "BurgYl")[7], size = 2) +  
  geom_point(aes(y = out),  
             color = carto_pal(7, "BurgYl")[5], size = 2) +  
  geom_errorbar(aes(x = n_par + .15,  
                    ymin = `in`, ymax = out),  
                width = .1, color = carto_pal(7, "BurgYl")[6]) +  
  geom_text(aes(x = n_par + .4,  
                y = (out + `in`) / 2,  
                label = height),  
            family = "Courier", size = 3, color = carto_pal(7, "BurgYl")[6]) +  
  labs(x = "number of parameters",  
       y = "deviance") +  
  theme_classic() +  
  theme(text = element_text(family = "Courier"),  
        strip.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[1], 1/4), color = "white"))
```

```
panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[3], 1/4))) +
facet_wrap(~n, scale = "free_y")
```



Again, our numbers aren't the exact same as McElreath's because a) this is a simulation and b) our number of simulations was an order of magnitude smaller than his. But the overall pattern is the same. More to the point, the distances between the in- and out-of-sample points

are nearly the same, for each model, at both $N = 20$ (left) and $N = 100$ (right). Each distance is nearly twice the number of parameters, as labeled on the horizontal axis. The dashed lines show exactly the [dark purple] points plus twice the number of parameters, tracing closely along the average out-of-sample deviance for each model.

This is the phenomenon behind information criteria. (p. 189)

In the text, McElreath focused on the DIC and WAIC. As you'll see, the LOO has increased in popularity since he published the text. Going forward, we'll juggle the WAIC and the LOO in this project. But we will respect the text and work in a little DIC talk.

6.4.1 DIC.

The DIC has been widely used for some time, now. For a great talk on the DIC, check out the authoritative David Spiegelhalter's [Retrospective read paper: Bayesian measure of model complexity and fit](#). If we define D as the deviance's posterior distribution, \bar{D} as its mean and \hat{D} as the deviance when computed at the posterior mean, then we define the DIC as

$$\text{DIC} = \bar{D} + (\bar{D} + \hat{D}) + \bar{D} + p_D$$

And p_D is the number of effective parameters in the model, which is also sometimes referred to as the penalty term. As you'll see, you can get the p_D for `brms::brm()` models. However, I'm not aware of a way to that `brms` or the `loo` package—to be introduced shortly—offer convenience functions that yield the DIC.

6.4.2 WAIC.

It's okay that the `brms` and `loo` packages don't yield the DIC because

even better than the DIC is the Widely Applicable Information Criterion (WAIC)...

Define $\Pr(y_i)$ as the average likelihood of observation i in the training sample. This means we compute the likelihood of y_i for each set of parameters sampled from the posterior distribution. Then we average the likelihoods for each observation i and finally sum over all observations. This produces the first part of WAIC, the log-pointwise-predictive-density, lppd:

$$\text{lppd} = \sum_{i=1}^N \log \Pr(y_i)$$

You might say this out loud as:

The log-pointwise-predictive-density is the total across observations of the logarithm of the average likelihood of each observation.

... The second piece of WAIC is the effect number of parameters p_{WAIC} . Define $V(y_i)$ as the variance in log-likelihood for observation i in the training sample. This means we compute the log-likelihood for observation y_i for each sample from the posterior distribution. Then we take the variance of those values. This is $V(y_i)$. Now p_{WAIC} is defined as:

$$p_{\text{WAIC}} = \sum_{i=1}^N V(y_i)$$

Now WAIC is defined as:

$$\text{WAIC} = -2(\text{lppd} - p_{\text{WAIC}})$$

And this value is yet another estimate of out-of-sample deviance. (pp. 191–192)

You'll see how to compute the WAIC in brms in just a bit.

6.4.2.1 Overthinking: WAIC calculation.

Here is how to fit the pre-WAIC model in brms.

```
data(cars)

b <-
  brm(data = cars, family = gaussian,
       dist ~ 1 + speed,
       prior = c(prior(normal(0, 100), class = Intercept),
                 prior(normal(0, 10), class = b),
                 prior(uniform(0, 30), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       seed = 6)
```

Here's the summary.

```
print(b)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: dist ~ 1 + speed
##   Data: cars (Number of observations: 50)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     -17.40      7.07   -30.97    -3.67        1920  1.00
## speed          3.92      0.44     3.07     4.75        1783  1.00
##
## Family Specific Parameters:
##               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
```

```
## sigma      15.83      1.69     12.98     19.47      2068 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

In brms, you return the loglikelihood with `log_lik()`.

```
ll <-
  b %>%
  log_lik() %>%
  as_tibble()
```

Computing the lppd, the “Bayesian deviance”, takes a bit of leg work.

```
dfmean <-
  ll %>%
  exp() %>%
  summarise_all(mean) %>%
  gather(key, means) %>%
  select(means) %>%
  log()

(
  lppd <-
  dfmean %>%
  sum()
)
```

```
## [1] -206.6836
```

Comupting the effective number of parameters, p_{WAIC} , isn’t much better.

```
dfvar <-
  ll %>%
  summarise_all(var) %>%
  gather(key, vars) %>%
  select(vars)

pwaic <-
  dfvar %>%
  sum()

pwaic
```

```
## [1] 3.418859
```

Finally, here’s what we’ve been working so hard for: our hand calculated WAIC value. Compare it to the value returned by the brms `waic()` function.

```
-2 * (lppd - pwaic)
```

```
## [1] 420.205
```

```
waic(b)
```

```
##  
## Computed from 4000 by 50 log-likelihood matrix  
##  
##           Estimate    SE  
## elpd_waic   -210.1  6.4  
## p_waic      3.4   1.2  
## waic        420.2 12.7
```

Before we move on, did you notice the `elpd_waic` row in the tibble `waic()` returned? That value is the `lppd` minus the `pwaic`, but without multiplying the result by -2. E.g.,

```
lppd - pwaic
```

```
## [1] -210.1025
```

That tidbit will come in handy a little bit later. But for now, here's how we compute the WAIC standard error.

```
dfmean %>%  
  mutate(waic_vec = -2 * (means - dfvar$vars)) %>%  
  summarise(waic_se = (var(waic_vec) * nrow(dfmean)) %>% sqrt())
```

```
## # A tibble: 1 x 1  
##   waic_se  
##     <dbl>  
## 1     12.7
```

6.4.3 DIC and WAIC as estimates of deviance.

Once again, we'll wrap McElreath's `sim.train.test()`-based simulation code within a custom function, `make_sim()`. This time we've adjusted `make_sim()` to take one argument, `b_sigma`. We will then feed that value into the same-named argument within `sim.train.test()`. Also notice that within `sim.train.test()`, we've specified TRUE for the information criteria and deviance arguments. Be warned: it takes extra time to compute the WAIC. Because we do that for every model, this simulation takes longer than the previous ones. To get a taste, try running it with something like `n_sim <- 5` first.

```
n_sim <- 1e3  
  
make_sim <- function(b_sigma){  
  sapply(kseq, function(k) {  
    print(k);  
    r <- mcreplicate(n_sim,  
      sim.train.test(N      = 20,  
                     k       = k,  
                     b_sigma = b_sigma,  
                     DIC     = T,  
                     WAIC    = T,  
                     devbar  = T,  
                     devbarout = T),  
      mc.cores = n_cores);  
  
    c(dev_in   = mean(r[1, ]),  
      dev_out  = mean(r[2, ]),  
      DIC      = mean(r[3, ]),  
      WAIC     = mean(r[4, ]),  
      devbar   = mean(r[5, ]),  
      devbarout = mean(r[6, ]))  
  })  
} %>%
```

```

data.frame() %>%
rownames_to_column() %>%
rename(statistic = rowname)
}

s <-
tibble(b_sigma = c(100, .5)) %>%
mutate(sim = purrr::map(b_sigma, make_sim)) %>%
unnest()

```

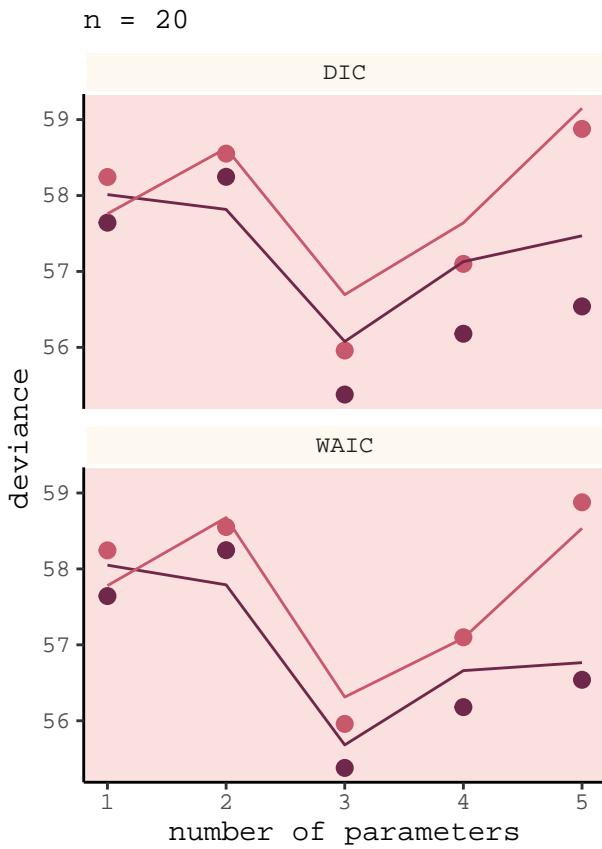
Here we wrangle and plot.

```

s %>%
gather(n_par, value, -b_sigma, -statistic) %>%
mutate(n_par = str_remove(n_par, "X") %>% as.double()) %>%
filter(statistic != "devbar" & statistic != "devbarout") %>%
spread(key = statistic, value = value) %>%
gather(ic, value, -b_sigma, -n_par, -dev_in, -dev_out) %>%
gather(sample, deviance, -b_sigma, -n_par, -ic, -value) %>%
filter(sample == "dev_out") %>%
mutate(b_sigma = b_sigma %>% as.character()) %>%

ggplot(aes(x = n_par)) +
geom_point(aes(y = deviance, color = b_sigma),
           size = 2.5) +
geom_line(aes(y = value, group = b_sigma, color = b_sigma)) +
scale_color_manual(values = c(carto_pal(7, "BurgYl")[7], carto_pal(7, "BurgYl")[5])) +
# scale_color_manual(values = c("steelblue", "black")) +
labs(subtitle = "n = 20",
     x = "number of parameters",
     y = "deviance") +
theme_classic() +
theme(text = element_text(family = "Courier"),
      strip.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[1], 1/4), color = "white"),
      panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)),
      legend.position = "none") +
facet_wrap(~ic, ncol = 1)

```



And again, our results don't perfectly match those in the text because a) we're simulating and b) we used fewer iterations than McElreath did. But the overall pattern remains.

6.5 Using information criteria

In contrast to model selection, “this section provides a brief example of model *comparison* and *averaging*” (p. 195, *emphasis* in the original).

6.5.1 Model comparison.

Load the `milk` data from earlier in the text.

```
library(rethinking)
data(milk)

d <-
  milk %>%
  drop_na(ends_with("_s"))
rm(milk)

d <-
  d %>%
  mutate(neocortex = neocortex.perc / 100)
```

The dimensions of `d` are:

```
dim(d)
```

```
## [1] 17 9
```

Load brms.

```
detach(package:rstan, unload = T)
library(brms)
```

We're ready to fit the competing `kcal.per.g` models. Note our use of `update()` in the last two models.

```
inits <- list(Intercept = mean(d$kcal.per.g),
              sigma      = sd(d$kcal.per.g))

inits_list <-list(inits, inits, inits)

b6.11 <-
  brm(data = d, family = gaussian,
       kcal.per.g ~ 1,
       prior = c(prior(uniform(-1000, 1000), class = Intercept),
                  prior(uniform(0, 100), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       inits = inits_list,
       seed = 6)

inits <- list(Intercept = mean(d$kcal.per.g),
              neocortex = 0,
              sigma      = sd(d$kcal.per.g))
inits_list <-list(inits, inits, inits)

b6.12 <-
  brm(data = d, family = gaussian,
       kcal.per.g ~ 1 + neocortex,
       prior = c(prior(uniform(-1000, 1000), class = Intercept),
                  prior(uniform(-1000, 1000), class = b),
                  prior(uniform(0, 100), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       inits = inits_list,
       seed = 6)

inits <- list(Intercept    = mean(d$kcal.per.g),
              `log(mass)` = 0,
              sigma        = sd(d$kcal.per.g))
inits_list <-list(inits, inits, inits)

b6.13 <-
  update(b6.12,
         newdata = d,
         formula = kcal.per.g ~ 1 + log(mass),
         inits   = inits_list)

inits <- list(Intercept    = mean(d$kcal.per.g),
              neocortex   = 0,
              `log(mass)` = 0,
              sigma        = sd(d$kcal.per.g))
inits_list <-list(inits, inits, inits)

b6.14 <-
  update(b6.13,
         newdata = d,
         formula = kcal.per.g ~ 1 + neocortex + log(mass),
         inits   = inits_list)
```

6.5.1.1 Comparing WAIC values.

In brms, you can get a model's WAIC value with the `waic()` function.

```
waic(b6.14)
```

```
##  
## Computed from 4000 by 17 log-likelihood matrix  
##  
##           Estimate   SE  
## elpd_waic     8.3 2.6  
## p_waic        3.2 0.9  
## waic       -16.5 5.2  
  
## Warning: 2 (11.8%) p_waic estimates greater than 0.4. We recommend trying loo instead.
```

Note the warning message. Statisticians have made notable advances in Bayesian information criteria since McElreath published *Statistical Rethinking*. I won't go into detail here, but the "We recommend trying loo instead" part of the message is designed to prompt us to use a different information criteria, the Pareto smoothed importance-sampling leave-one-out cross-validation (PSIS-LOO; aka, the LOO). In brms this is available with the `loo()` function, which you can learn more about in [this vignette](#) from the makers of the [loo package](#). For now, back to the WAIC.

There are a few ways to approach information criteria within the brms framework. If all you want are the quick results for a model, just plug the name of your `brm()` fit object into the `waic()` function.

```
waic(b6.11)
```

```
##  
## Computed from 4000 by 17 log-likelihood matrix  
##  
##           Estimate   SE  
## elpd_waic     4.4 1.9  
## p_waic        1.3 0.3  
## waic       -8.9 3.8
```

The WAIC and its standard error are on the bottom row. The p_{WAIC} and its SE are stacked atop that. And look there on the top row. Remember how we pointed out, above, that we get the WAIC by multiplying ($lppd - p_{WAIC}$) by -2 ? Well, if you just do the subtraction without multiplying the result by -2 , you get the `elpd_waic`. File that away. It'll become important in a bit.

Following the version 2.8.0 update, part of the suggested workflow for using information criteria with brms (i.e., execute `?loo.brmsfit`) is to add the estimates to the `brm()` fit object itself. You do that with the `add_criterion()` function. Here's how we'd do so with `b6.11`.

```
b6.11 <- add_criterion(b6.11, "waic")
```

With that in place, here's how you'd extract the WAIC information from the fit object.

```
b6.11$waic
```

```
##  
## Computed from 4000 by 17 log-likelihood matrix  
##  
##           Estimate   SE  
## elpd_waic     4.4 1.9  
## p_waic        1.3 0.3  
## waic       -8.9 3.8
```

Why would I go through all that trouble?, you might ask. Well, two reasons. First, now your WAIC information is saved with all the rest of your fit output, which can be convenient. But second, it sets you up to use the `loo_compare()` function to compare models by their information criteria. To get a sense of that workflow, here we use `add_criterion()` for the next three models. Then we'll use `loo_compare()`.

```
# compute and save the WAIC information for the next three models
b6.12 <- add_criterion(b6.12, "waic")
b6.13 <- add_criterion(b6.13, "waic")
b6.14 <- add_criterion(b6.14, "waic")

# compare the WAIC estimates
w <- loo_compare(b6.11, b6.12, b6.13, b6.14,
                  criterion = "waic")

print(w)

##      elpd_diff se_diff
## b6.14    0.0     0.0
## b6.11   -3.8     2.5
## b6.13   -3.8     1.8
## b6.12   -4.7     2.5
```

You don't have to save those results as an object like we just did with `w`. But that'll serve some pedagogical purposes in just a bit. So go with it. With respect to the output, notice the `elpd_diff` column and the adjacent `se_diff` column. Those are our WAIC differences. The models have been rank ordered from the lowest (i.e., `b6.14`) to the highest (i.e., `b6.12`). The scores listed are the differences of `b6.14` minus the comparison model. Since `b6.14` is the comparison model in the top row, the values are naturally 0 (i.e., $x - x = 0$). But now here's another critical thing to understand: Since the brms version 2.8.0 update, WAIC and LOO differences are no longer reported in the $-2 * x$ metric. Remember how we keep rehearsing that multiplying (`lppd - pwaic`) by -2 is a historic artifact associated with the frequentist chi-square test? We'll, the makers of the `loo` package aren't fans and they no longer support the conversion.

So here's the deal. The substantive interpretations of the differences presented in an `elpd_diff` metric will be the same as if presented in a WAIC metric. But if we want to compare our `elpd_diff` results to those in the text, we will have to multiply them by -2. And also, if we want the associated standard error in the proper metric, we'll need to multiply the `se_diff` column by 2. You wouldn't multiply by -2 because that would return a negative standard error, which would be silly. Here's a quick way to do so.

```
cbind(waic_diff = w[, 1] * -2,
      se        = w[, 2] * 2)

##      waic_diff      se
## b6.14  0.000000 0.000000
## b6.11  7.657211 5.013383
## b6.13  7.676240 3.599457
## b6.12  9.468233 5.061744
```

One more thing. On page 198, and on many other pages to follow in the text, McElreath used the `rethinking::compare()` function to return a rich table of information about the WAIC information for several models. If we're tricky, we can do something similar with `loo_compare`. To learn how, let's peer further into the structure of our `w` object.

```
str(w)

##  'compare.loo' num [1:4, 1:8] 0 -3.83 -3.84 -4.73 0 ...
##  - attr(*, "dimnames")=List of 2
##    ..$ : chr [1:4] "b6.14" "b6.11" "b6.13" "b6.12"
##    ..$ : chr [1:8] "elpd_diff" "se_diff" "elpd_waic" "se_elpd_waic" ...
```

When we used `print(w)`, a few code blocks earlier, it only returned two columns. It appears we actually have eight. We can see the full output with the `simplify = F` argument.

```
print(w, simplify = F)

##      elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic   se_waic
## b6.14    0.0     0.0     8.3      2.6     3.2     0.9    -16.5    5.2
## b6.11   -3.8     2.5     4.4      1.9     1.3     0.3    -8.9    3.8
## b6.13   -3.8     1.8     4.4      2.1     2.0     0.4    -8.9    4.2
## b6.12   -4.7     2.5     3.5      1.6     2.0     0.3    -7.1    3.2
```

The results are quite analogous to those from `rethinking::compare()`. Again, the difference estimates are in the metric of the elpd. But the interpretation is the same and we can convert them to the traditional information criteria metric with simple multiplication. As we'll see later, this basic workflow applies to the LOO, too.

If you want to get those WAIC weights, you can use the `brms::model_weights()` function like so:

```
model_weights(b6.11, b6.12, b6.13, b6.14,
              weights = "waic") %>%
  round(digits = 2)

## b6.11 b6.12 b6.13 b6.14
##  0.02  0.01  0.02  0.96
```

That last `round()` line was just to limit the decimal-place precision. If you really wanted to go through the trouble, you could make yourself a little table like this:

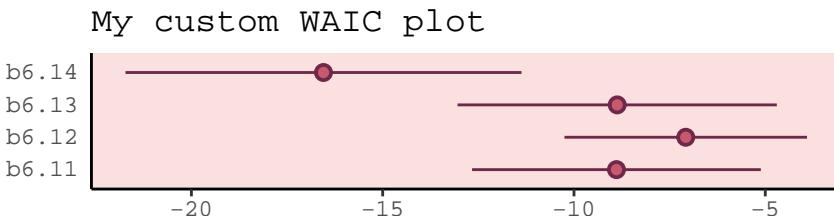
```
model_weights(b6.11, b6.12, b6.13, b6.14,
              weights = "waic") %>%
  as_tibble() %>%
  rename(weight = value) %>%
  mutate(model = c("b6.11", "b6.12", "b6.13", "b6.14"),
         weight = weight %>% round(digits = 2)) %>%
  select(model, weight) %>%
  arrange(desc(weight)) %>%
  knitr::kable()
```

model	weight
b6.14	0.96
b6.11	0.02
b6.13	0.02
b6.12	0.01

With a little [] subsetting and light wrangling, we can convert the contents of our `w` object to a format suitable for plotting the WAIC estimates.

```
w[, 7:8] %>%
  data.frame() %>%
  rownames_to_column(var = "model_name") %>%

  ggplot(aes(x = model_name,
             y = waic,
             ymin = waic - se_waic,
             ymax = waic + se_waic)) +
  geom_pointrange(shape = 21, color = carto_pal(7, "BurgYl")[7], fill = carto_pal(7, "BurgYl")[5]) +
  coord_flip() +
  labs(x = NULL, y = NULL,
       title = "My custom WAIC plot") +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        axis.ticks.y = element_blank(),
        panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[3], 1/4)))
```



We briefly discussed the alternative information criteria, the LOO, above. Here's how to use it in brms.

```
loo(b6.11)
```

```
##  
## Computed from 4000 by 17 log-likelihood matrix  
##  
##      Estimate   SE  
## elpd_loo     4.4 1.9  
## p_loo       1.3 0.3  
## looic     -8.8 3.8  
## -----  
## Monte Carlo SE of elpd_loo is 0.0.  
##  
## Pareto k diagnostic values:  
##                               Count Pct.    Min. n_eff  
## (-Inf, 0.5]   (good)      16  94.1%  3013  
## (0.5, 0.7]   (ok)        1  5.9%  2041  
## (0.7, 1]     (bad)       0  0.0% <NA>  
## (1, Inf)    (very bad)  0  0.0% <NA>  
##  
## All Pareto k estimates are ok (k < 0.7).  
## See help('pareto-k-diagnostic') for details.
```

The Pareto k values are a useful model fit diagnostic tool, which we'll discuss later. But for now, realize that brms uses functions from the [loo package](#) to compute its WAIC and LOO values. In addition to the vignette, above, [this vignette](#) demonstrates the LOO with these very same examples from McElreath's text. And if you'd like to dive a little deeper, check out [Aki Vehtari's GPSS2017 workshop](#) or his talk from November 2018, [Model assessment, selection and averaging](#).

Let's get back on track with the text. To put all this model comparison in perspective,

in this analysis, the best model has more than 90% of the model weight. That's pretty good. But with only 12 cases, the error on the WAIC estimate is substantial, and of course that uncertainty should propagate to the Akaike weights. So don't get too excited. If we take the standard error of the difference from the [`loo_compare()`] table literally, you can think of the difference as a Gaussian distribution centered (for the difference between models [b6.14 and b6.11]) on [9.47] with a standard deviation of [5.06]. (p. 200)

Here are those two values in the elpd metric.

```
w[4, 1:2]
```

```
## elpd_diff   se_diff  
## -4.734116  2.530872
```

And here we convert them to the WAIC metric.

```
round(w[4, 1] * -2, 2)
```

```
## [1] 9.47
```

```
round(w[4, 2] * 2, 2)
```

```
## [1] 5.06
```

If it's easier to see, here's the same information in a tibble.

```
tibble(value      = c("difference", "se"),
       elpd      = w[4, 1:2],
       conversion_factor = c(-2, 2)) %>%
     mutate(waic      = elpd * conversion_factor)
```

```
## # A tibble: 2 x 4
##   value      elpd conversion_factor waic
##   <chr>     <dbl>          <dbl>    <dbl>
## 1 difference -4.73           -2    9.47
## 2 se         2.53            2    5.06
```

Before we forget, McElreath gave some perspective difference between the models with the highest and lowest WAIC values (p. 200).

But to the point, we can extract the two numerals and plug them into `rnorm()`.

```
# how many draws would you like?
n <- 1e5

set.seed(6)

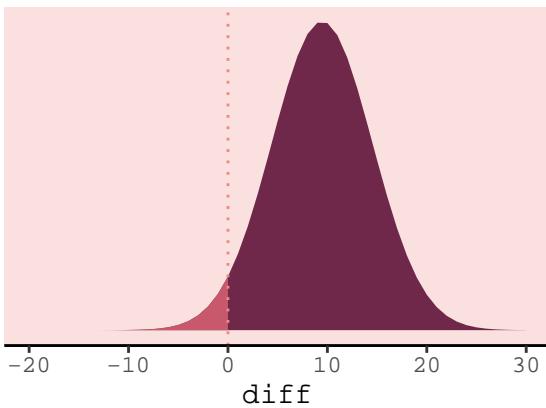
# simulate
diff <-
  tibble(diff = rnorm(n,
                      mean = w[4, 1] * -2,
                      sd    = w[4, 2] * 2))

diff %>%
  summarise(the_probability_a_difference_is_negative = sum(diff < 0) / n)
```

```
## # A tibble: 1 x 1
##   the_probability_a_difference_is_negative
##                     <dbl>
## 1                 0.0309
```

In case you're curious, this is a graphic version of what we just did.

```
tibble(diff = -20:30) %>%
  ggplot(aes(x = diff, ymin = 0)) +
  geom_ribbon(aes(ymax = dnorm(diff, w[4, 1] * -2, w[4, 2] * 2)),
              fill = carto_pal(7, "BurgYl")[7]) +
  geom_ribbon(data = tibble(diff = -20:0),
              aes(ymax = dnorm(diff, w[4, 1] * -2, w[4, 2] * 2)),
              fill = carto_pal(7, "BurgYl")[5]) +
  geom_vline(xintercept = 0, linetype = 3,
             color = carto_pal(7, "BurgYl")[3]) +
  scale_y_continuous(NULL, breaks = NULL) +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[3], 1/4)))
```



6.5.1.2 Comparing estimates.

The `brms` package doesn't have anything like `rethinking`'s `coeftab()` function. However, one can get that information with a little ingenuity. Here we'll employ the `broom::tidy()` function, which will save the summary statistics for our model parameters. For example, this is what it will produce for the full model, `b6.14`.

```
tidy(b6.14)
```

```
## # A tibble: 5 x 6
##   term      estimate std.error    lower    upper
## 1 b_Intercept -1.09554566 0.59579842 -2.0834891 -0.13210946
## 2 b_neocortex  2.80996247 0.92458417  1.2962875  4.33896238
## 3 b_logmass   -0.09679801 0.02814521 -0.1412482 -0.05140763
## 4 sigma       0.13984670 0.02999699  0.0996455  0.19499069
## 5 lp__        -19.24006410 1.60889794 -22.3871835 -17.33233588
```

Note, `tidy()` also grabs the log posterior (i.e., `lp__`), which we'll exclude for our purposes. With a little `purrr::map()` code, you can save the `brm()` fits and their `tidy()` summaries into a nested tibble, and then `unnest()` the tibble for `coeftab()`-like use.

```
my_coef_tab <-
  tibble(model = c("b6.11", "b6.12", "b6.13", "b6.14")) %>%
  mutate(fit = purrr::map(model, get)) %>%
  mutate(tidy = purrr::map(fit, tidy)) %>%
  unnest(tidy) %>%
  filter(term != "lp__")

head(my_coef_tab)
```

```
## # A tibble: 6 x 6
##   model term      estimate std.error    lower    upper
##   <chr> <chr>      <dbl>     <dbl>    <dbl>    <dbl>
## 1 b6.11 b_Intercept  0.656     0.0449   0.582   0.731
## 2 b6.11 sigma       0.187     0.0369   0.138   0.254
## 3 b6.12 b_Intercept  0.348     0.567   -0.579   1.27
## 4 b6.12 b_neocortex  0.460     0.834   -0.883   1.82
## 5 b6.12 sigma       0.194     0.0403   0.142   0.268
## 6 b6.13 b_Intercept  0.704     0.0575   0.612   0.796
```

Just a little more work and we'll have a table analogous to the one McElreath produced with his `coef_tab()` function.

```
my_coef_tab %>%
  # learn more about `dplyr::complete()` here: https://rdrr.io/cran/tidyr/man/expand.html
  complete(term = distinct(., term), model) %>%
  select(model, term, estimate) %>%
  mutate(estimate = round(estimate, digits = 2)) %>%
  spread(key = model, value = estimate)
```

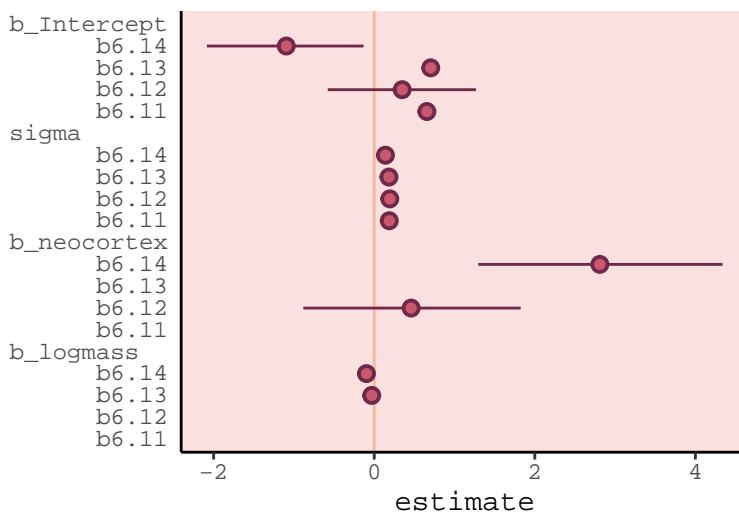
```
## # A tibble: 4 x 5
##   term      b6.11 b6.12 b6.13 b6.14
##   <chr>     <dbl> <dbl> <dbl> <dbl>
## 1 b_Intercept  0.66  0.35  0.7  -1.1
## 2 b_logmass    NA     NA    -0.03 -0.1
## 3 b_neocortex  NA     0.46  NA     2.81
## 4 sigma       0.19  0.19   0.18  0.14
```

I'm also not aware of an efficient way in brms to reproduce Figure 6.12 for which McElreath nested his `coeftab()` argument in a `plot()` argument. However, one can build something similar by hand with a little data wrangling.

```
# data wrangling
wrangled_my_coef_tab <-
  my_coef_tab %>%
  complete(term = distinct(., term), model) %>%
  rbind(
    tibble(
      model      = NA,
      term       = c("b_logmass", "b_neocortex", "sigma", "b_Intercept"),
      estimate   = NA,
      std.error  = NA,
      lower      = NA,
      upper      = NA)) %>%
  mutate(axis = ifelse(is.na(model), term, model),
         model = factor(model, levels = c("b6.11", "b6.12", "b6.13", "b6.14")),
         term  = factor(term, levels = c("b_logmass", "b_neocortex", "sigma", "b_Intercept", NA))) %>%
  arrange(term, model) %>%
  mutate(axis_order = letters[1:20],
        axis = ifelse(str_detect(axis, "b6."), str_c(" ", axis), axis))

# plot
ggplot(data = wrangled_my_coef_tab,
       aes(x = axis_order,
           y = estimate,
           ymin = lower,
           ymax = upper)) +
  theme_classic() +
  geom_hline(yintercept = 0, color = carto_pal(7, "BurgYl")[2]) +
  geom_pointrange(shape = 21, color = carto_pal(7, "BurgYl")[7], fill = carto_pal(7, "BurgYl")[5]) +
  scale_x_discrete(NULL, labels = wrangled_my_coef_tab$axis) +
  ggtitle("My other coeftab() plot") +
  coord_flip() +
  theme(text      = element_text(family = "Courier"),
        panel.grid = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))
```

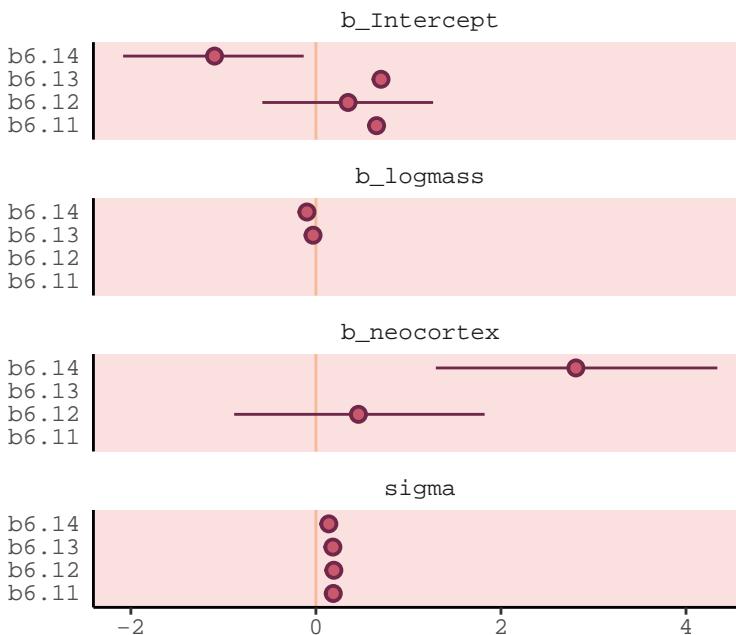
My other coeftab() plot



However, if you're willing to deviate just a bit from the format of McElreath's `coeftab()` plot, here's a more elegant way to work with our `my_coef_tab` tibble.

```
my_coef_tab %>%
```

```
ggplot(aes(x = model, y = estimate, ymin = lower, ymax = upper)) +
  geom_hline(yintercept = 0, color = carto_pal(7, "BurgYl")[2]) +
  geom_pointrange(shape = 21, color = carto_pal(7, "BurgYl")[7], fill = carto_pal(7, "BurgYl")[5]) +
  labs(x = NULL,
       y = NULL) +
  coord_flip() +
  theme_classic() +
  theme(text      = element_text(family = "Courier"),
        panel.grid = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)),
        strip.background = element_rect(color = "transparent")) +
  facet_wrap(~term, ncol = 1)
```



6.5.1.3 Rethinking: Barplots suck.

Man, I agree. “The only problem with barplots is that they have bars” (p. 203). You can find alternatives [here](#), [here](#), [here](#), and a whole bunch [here](#).

6.5.2 Model averaging.

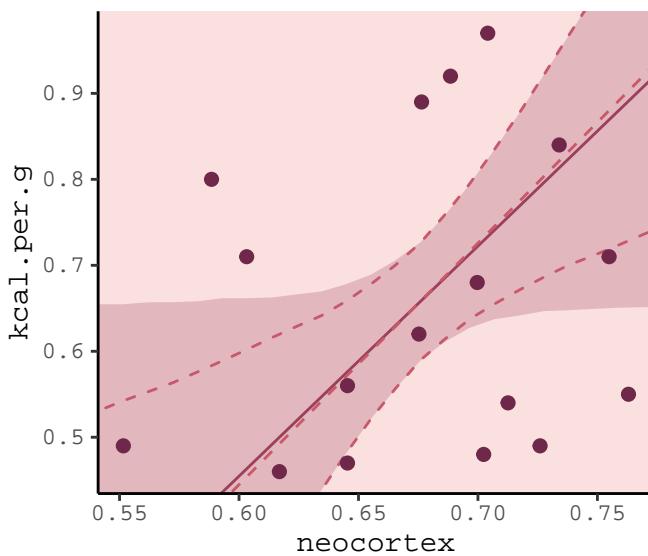
Within the current brms framework, you can do model-averaged predictions with the `pp_average()` function. The default weighting scheme is with the LOO. Here we’ll use the `weights = "waic"` argument to match McElreath’s method in the text. Because `pp_average()` yields a matrix, we’ll want to convert it to a tibble before feeding it into `ggplot2`.

```
# we need new data for both the `fitted()` and `pp_average()` functions
nd <-
  tibble(neocortex = seq(from = .5, to = .8, length.out = 30),
         mass       = 4.5)

# we'll get the `b6.14`-implied trajectory with `fitted()`
f <-
  fitted(b6.14, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

# the model-average trajectory comes from `pp_average()`
pp_average(b6.11, b6.12, b6.13, b6.14,
            weights = "waic",
            method   = "fitted", # for new data predictions, use `method = "predict"`
            newdata  = nd) %>%
  as_tibble() %>%
  bind_cols(nd) %>%

# plot Figure 6.13
ggplot(aes(x = neocortex, y = Estimate)) +
  geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5),
              fill   = carto_pal(7, "BurgYl")[6], alpha = 1/4) +
  geom_line(color = carto_pal(7, "BurgYl")[[6]]) +
  geom_ribbon(data  = f, aes(ymin = Q2.5, ymax = Q97.5),
              color = carto_pal(7, "BurgYl")[[5]], fill = "transparent", linetype = 2) +
  geom_line(data = f,
            color = carto_pal(7, "BurgYl")[[5]], linetype = 2) +
  geom_point(data = d, aes(y = kcal.per.g),
             size = 2, color = carto_pal(7, "BurgYl")[[7]]) +
  labs(y = "kcal.per.g") +
  coord_cartesian(xlim = range(d$neocortex),
                  ylim = range(d$kcal.per.g)) +
  theme_classic() +
  theme(text           = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(cartoon_pal(7, "BurgYl")[[3]], 1/4)))
```



6.6 Summary Bonus: R^2 talk

At the beginning of the chapter (pp. 167–168), McElreath briefly introduced R^2 as a popular way to assess the variance explained in a model. He pooh-poohed it because of its tendency to overfit. It's also limited in that it doesn't generalize well outside of the single-level Gaussian framework. However, if you should find yourself in a situation where R^2 suits your purposes, the brms `bayes_R2()` function might be of use. Simply feeding a model `brm` fit object into `bayes_R2()` will return the posterior mean, SD , and 95% intervals. For example:

```
bayes_R2(b6.14) %>% round(digits = 3)
```

```
##      Estimate Est.Error Q2.5 Q97.5
## R2      0.501     0.13  0.186  0.666
```

With just a little data processing, you can get a tibble table of each of models' R^2 'Estimate'.

```
rbind(bayes_R2(b6.11),
      bayes_R2(b6.12),
      bayes_R2(b6.13),
      bayes_R2(b6.14)) %>%
  as_tibble() %>%
  mutate(model = c("b6.11", "b6.12", "b6.13", "b6.14"),
         r_square_posterior_mean = round(Estimate, digits = 2)) %>%
  select(model, r_square_posterior_mean)

## # A tibble: 4 x 2
##   model r_square_posterior_mean
##   <chr>             <dbl>
## 1 b6.11              0
## 2 b6.12              0.08
## 3 b6.13              0.14
## 4 b6.14              0.5
```

If you want the full distribution of the R^2 , you'll need to add a `summary = F` argument. Note how this returns a numeric vector.

```
r2_b6.13 <- bayes_R2(b6.13, summary = F)

r2_b6.13 %>%
  glimpse()
```

```
##  num [1:4000, 1] 0.2245 0.0146 0.0808 0.288 0.1127 ...
## - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr "R2"
```

If you want to use these in ggplot2, you'll need to put them in tibbles or data frames. Here we do so for two of our model fits.

```
# model `b6.13`
r2_b6.13 <-
  bayes_R2(b6.13, summary = F) %>%
  as_tibble() %>%
  rename(r2_13 = R2)

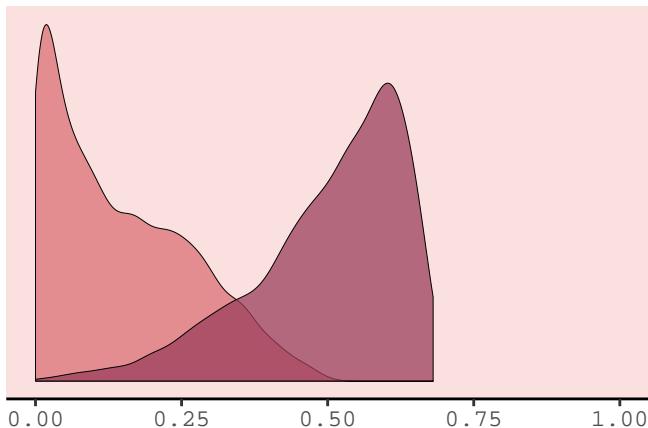
# model `b6.14`
r2_b6.14 <-
  bayes_R2(b6.14, summary = F) %>%
  as_tibble() %>%
  rename(r2_14 = R2)

# let's put them in the same data object
r2_combined <-
  bind_cols(r2_b6.13, r2_b6.14) %>%
  mutate(dif = r2_14 - r2_13)

# plot their densities
r2_combined %>%
  ggplot() +
  geom_density(aes(x = r2_13),
               fill = carto_pal(7, "BurgYl")[4], alpha = 3/4, size = 0, ) +
  geom_density(aes(x = r2_14),
               fill = carto_pal(7, "BurgYl")[6], alpha = 3/4, size = 0, ) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:1) +
  labs(x      = NULL,
       title   = expression(paste(italic("R")^{2}, " distributions")),
       subtitle = "Going from left to right, these are\nfor models b6.13 and b6.14.") +
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))
```

R^2 distributions

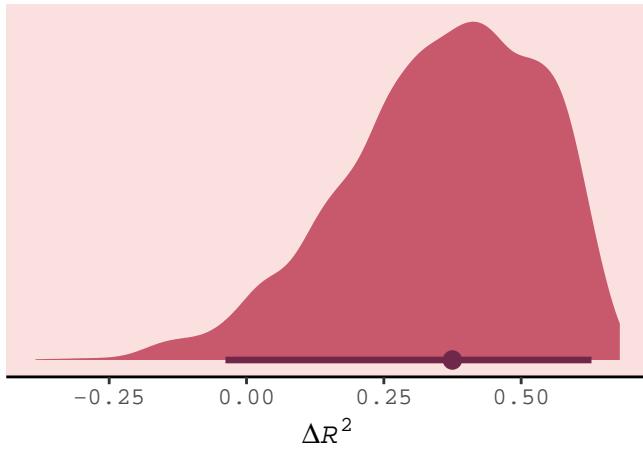
Going from left to right, these are
for models b6.13 and b6.14.



If you do your work in a field where folks use R^2 change, you might do that with a simple difference score, which we computed above with `mutate(dif = R2.14 - R2.13)`. Here's the ΔR^2 (i.e., `dif`) plot:

```
r2_combined %>%
  ggplot(aes(x = dif, y = 0)) +
  geom_halfeyeh(fill = carto_pal(7, "BurgYl")[5],
                 color = carto_pal(7, "BurgYl")[7],
                 point_interval = median_qi, .width = .95) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(x = expression(paste(Delta, italic("R")^2))),
  subtitle = "This is how much more variance, in\nterms of %, model b6.14 explained\ncompared to model b6.13.",
  theme_classic() +
  theme(text = element_text(family = "Courier"),
        panel.background = element_rect(fill = alpha(carto_pal(7, "BurgYl")[3], 1/4)))
```

This is how much more variance, in terms of %, model b6.14 explained compared to model b6.13.



The brms package did not get these R^2 values by traditional method used in, say, ordinary least squares estimation. To learn more about how the Bayesian R^2 sausage is made, check out the paper by [Gelman, Goodrich, Gabry, and Vehtari](#).

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods   base
##
```

```
## other attached packages:
## [1] rstan_2.18.2           StanHeaders_2.18.0-1 tidybayes_1.0.4      brms_2.8.8
## [5] Rcpp_1.0.1              gridExtra_2.3        broom_0.5.1          ggrepel_0.8.0
## [9] rcartocolor_1.0.0      forcats_0.3.0       stringr_1.4.0       dplyr_0.8.0.1
## [13] purrrr_0.2.5           readr_1.1.1         tidyverse_1.2.1     tibble_2.1.1
## [17] ggplot2_3.1.1          tidyverse_1.2.1

## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2        ggridges_0.5.0      rsconnect_0.8.8
## [4] rprojroot_1.3-2        ggstance_0.3       markdown_0.8
## [7] base64enc_0.1-3        rstudioapi_0.7     svUnit_0.7-12
## [10] DT_0.4                 fansi_0.4.0        mvtnorm_1.0-10
## [13] lubridate_1.7.4        xml2_1.2.0         codetools_0.2-15
## [16] bridgesampling_0.6-0   knitr_1.20         shinythemes_1.1.1
## [19] bayesplot_1.6.0        jsonlite_1.5       shiny_1.1.0
## [22] compiler_3.5.1         httr_1.3.1         backports_1.1.4
## [25] assertthat_0.2.0       Matrix_1.2-14      lazyeval_0.2.2
## [28] cli_1.0.1              later_0.7.3        htmltools_0.3.6
## [31] prettyunits_1.0.2      tools_3.5.1        igraph_1.2.1
## [34] coda_0.19-2            gtable_0.3.0       glue_1.3.1.9000
## [37] reshape2_1.4.3          cellranger_1.1.0   nlme_3.1-137
## [40] crosstalk_1.0.0        xfun_0.3           ps_1.2.1
## [43] rvest_0.3.2             mime_0.5           miniUI_0.1.1.1
## [46] gtools_3.8.1            MASS_7.3-50         zoo_1.8-2
## [49] scales_1.0.0            colourpicker_1.0   hms_0.4.2
## [52] promises_1.0.1          Brobdingnag_1.2-6  inline_0.3.15
## [55] shinystan_2.5.0          yaml_2.1.19        loo_2.1.0
## [58] stringi_1.4.3           dygraphs_1.1.1.5  pkgbuild_1.0.2
## [61] rlang_0.3.4              pkgconfig_2.0.2    matrixStats_0.54.0
## [64] evaluate_0.10.1          lattice_0.20-35   rstantools_1.5.1
## [67] htmlwidgets_1.2          labeling_0.3       tidyselect_0.2.5
## [70] processx_3.2.1          plyr_1.8.4         magrittr_1.5
## [73] bookdown_0.9             R6_2.3.0           generics_0.0.2
## [76] pillar_1.3.1             haven_1.1.2        withr_2.1.2
## [79] xts_0.10-2               abind_1.4-5        modelr_0.1.2
## [82] crayon_1.3.4             arrayhelpers_1.0-20160527 utf8_1.1.4
## [85] rmarkdown_1.10            grid_3.5.1         readxl_1.1.0
## [88] callr_3.1.0              threejs_0.3.1      digest_0.6.18
## [91] xtable_1.8-2              httpuv_1.4.4.2    stats4_3.5.1
## [94] munsell_0.5.0            viridisLite_0.3.0  shinyjs_1.0
```


Chapter 7

Interactions

Every model so far in [McElreath’s text] has assumed that each predictor has an independent association with the mean of the outcome. What if we want to allow the association to be conditional?... To model deeper conditionality—where the importance of one predictor depends upon another predictor—we need interaction. Interaction is a kind of conditioning, a way of allowing parameters (really their posterior distributions) to be conditional on further aspects of the data. (p. 210)

7.1 Building an interaction.

“Africa is special” (p. 211). Let’s load the `rugged` data to see one of the reasons why.

```
library(rethinking)

## Warning: package 'ggplot2' was built under R version 3.5.2

data(rugged)
d <- rugged
```

And here we switch out `rethinking` for `brms`.

```
detach(package:rethinking, unload = T)
library(brms)
rm(rugged)
```

We’ll continue to use tidyverse-style syntax to wrangle the data.

```
library(tidyverse)

# make the log version of criterion
d <-
  d %>%
  mutate(log_gdp = log(rgdppc_2000))

# extract countries with GDP data
dd <-
  d %>%
  filter(complete.cases(rgdppc_2000))

# split the data into countries in Africa and not in Africa
d.A1 <-
  dd %>%
```

```
filter(cont_africa == 1)

d.A0 <-
  dd %>%
  filter(cont_africa == 0)
```

The first two models predicting `log_gdp` are univariable.

```
b7.1 <-
  brm(data = d.A1, family = gaussian,
    log_gdp ~ 1 + rugged,
    prior = c(prior(normal(8, 100), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(uniform(0, 10), class = sigma)),
    iter = 2000, warmup = 1000, chains = 4, cores = 4,
    seed = 7)

b7.2 <-
  update(b7.1,
    newdata = d.A0)
```

In the text, McElreath more or less dared us to figure out how to make Figure 7.2. Here's the brms-relevant data wrangling.

```
nd <-
  tibble(rugged = seq(from = 0, to = 6.3, length.out = 30))

f_b7.1 <-
  fitted(b7.1, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

f_b7.2 <-
  fitted(b7.2, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

# here we'll put both in a single data object, with `f_b7.1` stacked atop `f_b7.2`
f <-
  full_join(f_b7.1, f_b7.2) %>%
  mutate(cont_africa = rep(c("Africa", "not Africa"), each = 30))
```

For this chapter, we'll take our plot theme from the `ggthemes` package.

```
# install.packages("ggthemes", dependencies = T)
library(ggthemes)
```

Here's the plot code for our version of Figure 7.2.

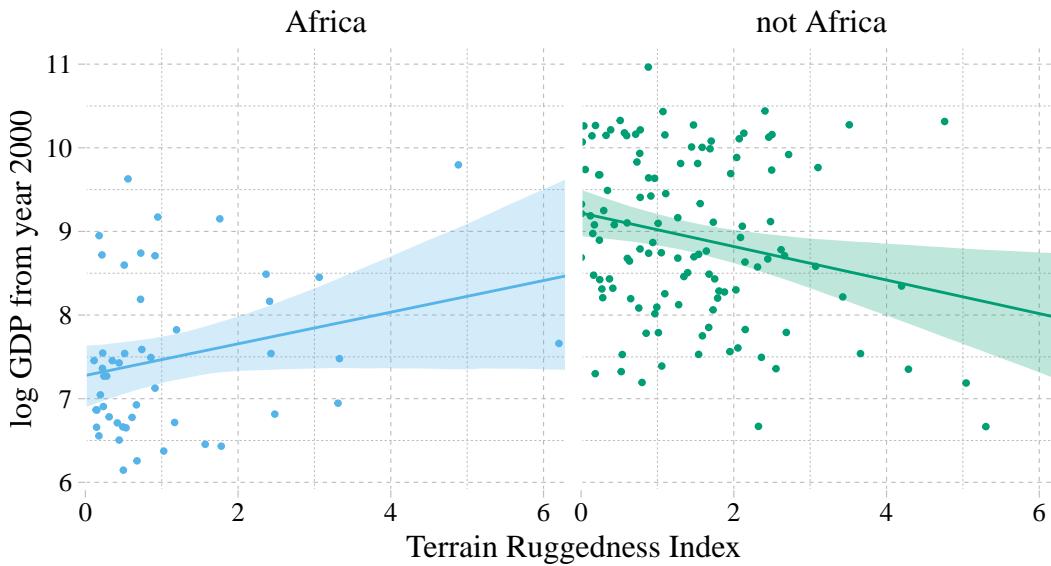
```
dd %>%
  mutate(cont_africa = ifelse(cont_africa == 1, "Africa", "not Africa")) %>%

  ggplot(aes(x = rugged)) +
  geom_smooth(data = f,
    aes(y = Estimate, ymin = Q2.5, ymax = Q97.5,
        fill = cont_africa, color = cont_africa),
    stat = "identity",
    alpha = 1/4, size = 1/2) +
  geom_point(aes(y = log_gdp, color = cont_africa),
```

```

size = 2/3) +
scale_colour_pander() +
scale_fill_pander() +
scale_x_continuous("Terrain Ruggedness Index", expand = c(0, 0)) +
ylab("log GDP from year 2000") +
theme_pander() +
theme(text = element_text(family = "Times")),
legend.position = "none") +
facet_wrap(~cont_africa)

```



7.1.1 Adding a dummy variable doesn't work.

Here's our model with all the countries, but without the `cont_africa` dummy.

```

b7.3 <-
  update(b7.1,
    newdata = dd)

```

Now we'll add the dummy.

```

b7.4 <-
  update(b7.3,
    newdata = dd,
    formula = log_gdp ~ 1 + rugged + cont_africa)

```

Using the skills from Chapter 6, let's compute the information criteria for the two models. Note how with the `add_criterion()` function, you can compute both the LOO and the WAIC at once.

```

b7.3 <- add_criterion(b7.3, c("loo", "waic"))
b7.4 <- add_criterion(b7.4, c("loo", "waic"))

```

Here we'll compare the models with the `loo_compare()` function, first by the WAIC and then by the LOO.

```

loo_compare(b7.3, b7.4,
            criterion = "waic")

```

```

##      elpd_diff se_diff
## b7.4    0.0     0.0
## b7.3   -31.6    7.3

```

```
loo_compare(b7.3, b7.4,
            criterion = "loo")

##      elpd_diff se_diff
## b7.4    0.0      0.0
## b7.3 -31.6     7.3
```

Happily, the WAIC and the LOO are in agreement. The model with the dummy, `b7.4`, fit the data much better. Here are the WAIC model weights.

```
model_weights(b7.3, b7.4,
              weights = "waic") %>%
  round(digits = 3)

## b7.3 b7.4
##   0     1
```

As in the text, almost all the weight went to the multivariable model, `b7.4`. Before we can plot that model, we need to wrangle a bit.

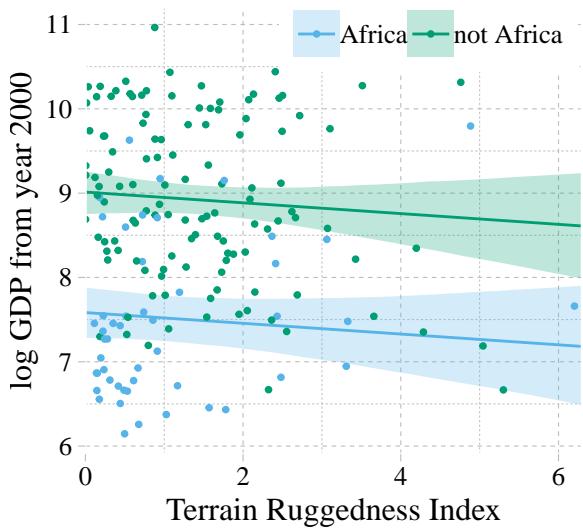
```
nd <- tibble(rugged      = seq(from = 0, to = 6.3, length.out = 30) %>%
  rep(., times = 2),
  cont_africa = rep(0:1, each = 30))

f <- fitted(b7.4, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd) %>%
  mutate(cont_africa = ifelse(cont_africa == 1, "Africa", "not Africa"))
```

Behold our Figure 7.3.

```
dd %>%
  mutate(cont_africa = ifelse(cont_africa == 1, "Africa", "not Africa")) %>%

ggplot(aes(x = rugged)) +
  geom_smooth(data = f,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5,
                  fill = cont_africa, color = cont_africa),
              stat = "identity",
              alpha = 1/4, size = 1/2) +
  geom_point(aes(y = log_gdp, color = cont_africa),
              size = 2/3) +
  scale_colour_pander() +
  scale_fill_pander() +
  scale_x_continuous("Terrain Ruggedness Index", expand = c(0, 0)) +
  ylab("log GDP from year 2000") +
  theme_pander() +
  theme(text = element_text(family = "Times"),
        legend.position = c(.69, .94),
        legend.title = element_blank(),
        legend.direction = "horizontal")
```



7.1.2 Adding a linear interaction does work.

Yes, it sure does. But before we fit, here's the equation:

$$\begin{aligned}\log_{\text{gdp}}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \gamma_i \text{rugged}_i + \beta_2 \text{cont_africa}_i \\ \gamma_i &= \beta_1 + \beta_3 \text{cont_africa}_i\end{aligned}$$

Fit the model.

```
b7.5 <-  
  update(b7.4,  
         formula = log_gdp ~ 1 + rugged*cont_africa)
```

For kicks, we'll just use the LOO to compare the last three models.

```
b7.5 <- add_criterion(b7.5, c("loo", "waic"))  
  
l <- loo_compare(b7.3, b7.4, b7.5,  
                  criterion = "loo")  
  
print(l, simplify = F)  
  
##      elpd_diff se_diff elpd_loo se_elpd_loo p_loo    se_p_loo looic   se_looic  
## b7.5      0.0      0.0 -234.7     7.3      5.1      0.9 469.4    14.6  
## b7.4     -3.3      3.0 -238.0     7.4      4.2      0.8 476.1    14.8  
## b7.3    -34.9     7.4 -269.6     6.5      2.5      0.3 539.3    13.0
```

And recall, if we want those LOO difference scores in the traditional metric like McElreath displayed in the text, we can do a quick conversion with algebra and `cbind()`.

```
cbind(loo_diff = l[, 1] * -2,  
      se       = l[, 2] * 2)  
  
##      loo_diff      se  
## b7.5  0.000000  0.000000  
## b7.4  6.657423  5.991091  
## b7.3 69.843359 14.710652
```

And we can weight the models based on the LOO rather than the WAIC, too.

```
model_weights(b7.3, b7.4, b7.5,
              weights = "loo") %>%
  round(digits = 3)
```

```
## b7.3  b7.4  b7.5
## 0.000  0.035  0.965
```

7.1.2.1 Overthinking: Conventional form of interaction.

The conventional equation for the interaction model might look like:

$$\log_{\text{gdp}}_i \sim \text{Normal}(\mu_i, \sigma)$$

$$\mu_i = \alpha + \beta_1 \text{rugged}_i + \beta_2 \text{cont_africa}_i + \beta_3 \text{rugged}_i \times \text{cont_africa}_i$$

Instead of the `y ~ 1 + x1*x2` approach, which will work fine with `brm()`, you can use this more explicit syntax.

```
b7.5b <-
  update(b7.5,
         formula = log_gdp ~ 1 + rugged + cont_africa + rugged:cont_africa)
```

From here on, I will default to this style of syntax for interactions.

Since this is the same model, it yields the same information criteria estimates within simulation error. Here we'll confirm that with the LOO.

```
b7.5b <- add_criterion(b7.5b, c("loo", "waic"))
```

```
b7.5b$loo
```

```
##
## Computed from 4000 by 170 log-likelihood matrix
##
##           Estimate    SE
## elpd_loo   -234.7  7.3
## p_loo       5.1   0.9
## looic      469.4 14.6
## -----
## Monte Carlo SE of elpd_loo is 0.0.
##
## Pareto k diagnostic values:
##                               Count Pct. Min. n_eff
## (-Inf, 0.5]   (good)     169   99.4%  1026
## (0.5, 0.7]   (ok)        1   0.6%  1672
## (0.7, 1]     (bad)       0   0.0%  <NA>
## (1, Inf)    (very bad)  0   0.0%  <NA>
##
## All Pareto k estimates are ok (k < 0.7).
## See help('pareto-k-diagnostic') for details.
```

```
b7.5b$loo
```

```
##
## Computed from 4000 by 170 log-likelihood matrix
##
```

```

##             Estimate    SE
## elpd_loo     -234.7  7.3
## p_loo        5.0   0.9
## looic       469.5 14.6
## -----
## Monte Carlo SE of elpd_loo is 0.1.
##
## Pareto k diagnostic values:
##                               Count Pct. Min. n_eff
## (-Inf, 0.5]    (good)    168  98.8%  2418
## (0.5, 0.7]    (ok)      2   1.2%  927
## (0.7, 1]      (bad)     0   0.0% <NA>
## (1, Inf)     (very bad) 0   0.0% <NA>
##
## All Pareto k estimates are ok (k < 0.7).
## See help('pareto-k-diagnostic') for details.

```

7.1.3 Plotting the interaction.

Here's our prep work for the figure.

```

f <-
  fitted(b7.5, newdata = nd) %>% # we can use the same `nd` data from last time
  as_tibble() %>%
  bind_cols(nd) %>%
  mutate(cont_africa = ifelse(cont_africa == 1, "Africa", "not Africa"))

```

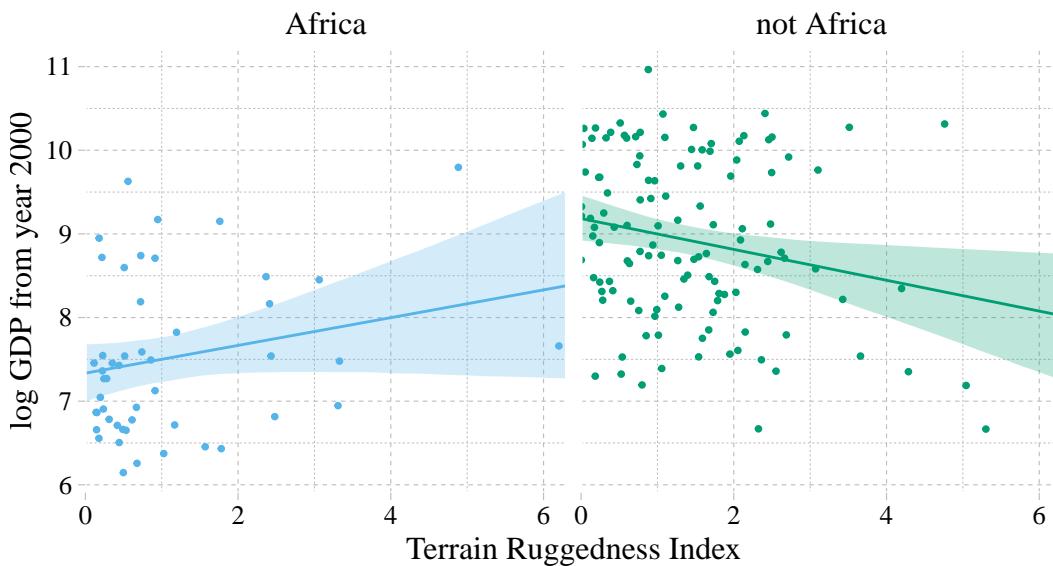
And here's the code for our version of Figure 7.4.

```

dd %>%
  mutate(cont_africa = ifelse(cont_africa == 1, "Africa", "not Africa")) %>%

  ggplot(aes(x = rugged, color = cont_africa)) +
  geom_smooth(data = f,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5,
                  fill = cont_africa),
              stat = "identity",
              alpha = 1/4, size = 1/2) +
  geom_point(aes(y = log_gdp),
              size = 2/3) +
  scale_colour_pander() +
  scale_fill_pander() +
  scale_x_continuous("Terrain Ruggedness Index", expand = c(0, 0)) +
  ylab("log GDP from year 2000") +
  theme_pander() +
  theme(text = element_text(family = "Times"),
        legend.position = "none") +
  facet_wrap(~cont_africa)

```



7.1.4 Interpreting an interaction estimate.

Interpreting interaction estimates is tricky. It's trickier than interpreting ordinary estimates. And for this reason, I usually advise against trying to understand an interaction from tables of numbers alone. Plotting implied predictions does far more for both our own understanding and for our audience's. (p. 219)

7.1.4.1 Parameters change meaning.

In a simple linear regression with no interactions, each coefficient says how much the average outcome, μ , changes when the predictor changes by one unit. And since all of the parameters have independent influences on the outcome, there's no trouble in interpreting each parameter separately. Each slope parameter gives us a direct measure of each predictor variable's influence.

Interaction models ruin this paradise. (p. 220)

Return the parameter estimates.

```
posterior_summary(b7.5)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	9.1833793	0.13748457	8.9217550	9.4610077
## b_rugged	-0.1844133	0.07571642	-0.3373420	-0.0362456
## b_cont_africa	-1.8485996	0.21877561	-2.2697449	-1.4312849
## b_rugged:cont_africa	0.3503760	0.12675316	0.1065656	0.5977352
## sigma	0.9508023	0.05333902	0.8558372	1.0608537
## lp__	-244.4268107	1.58112334	-248.3984388	-242.3406116

"Since γ (gamma) doesn't appear in this table—it wasn't estimated—we have to compute it ourselves" (p. 221). Like in the text, we'll do so first by working with the point estimates.

```
# within Africa
fixef(b7.5)[2, 1] + fixef(b7.5)[4, 1] * 1
```

```
## [1] 0.1659627
```

```
# outside Africa
fixef(b7.5)[2, 1] + fixef(b7.5)[4, 1] * 0
```

```
## [1] -0.1844133
```

7.1.4.2 Incorporating uncertainty.

To get some idea of the uncertainty around those γ values, we'll need to use the whole posterior. Since γ depends upon parameters, and those parameters have a posterior distribution, γ must also have a posterior distribution. Read the previous sentence again a few times. It's one of the most important concepts in processing Bayesian model fits. Anything calculated using parameters has a distribution. (p. 212)

Like McElreath, we'll avoid integral calculus in favor of working with the `posterior_samples()`.

```
post <- posterior_samples(b7.5)

post %>%
  transmute(gamma_Africa      = b_rugged + `b_rugged:cont_africa`,
            gamma_notAfrica = b_rugged) %>%
  gather(key, value) %>%
  group_by(key) %>%
  summarise(mean = mean(value))
```

```
## # A tibble: 2 x 2
##   key           mean
##   <chr>        <dbl>
## 1 gamma_Africa  0.166
## 2 gamma_notAfrica -0.184
```

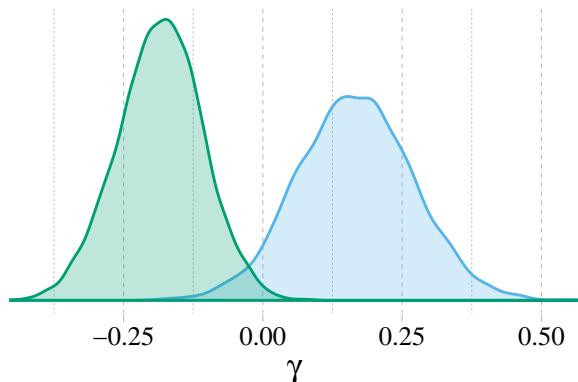
And here is our version of Figure 7.5.

```
post %>%
  transmute(gamma_Africa      = b_rugged + `b_rugged:cont_africa`,
            gamma_notAfrica = b_rugged) %>%
  gather(key, value) %>%

  ggplot(aes(x = value, group = key, color = key, fill = key)) +
  geom_density(alpha = 1/4) +
  scale_colour_pander() +
  scale_fill_pander() +
  scale_x_continuous(expression(gamma), expand = c(0, 0)) +
  scale_y_continuous(NULL, breaks = NULL) +
  ggtitle("Terraine Ruggedness slopes",
          subtitle = "Blue = African nations, Green = others") +
  theme_pander() +
  theme(text = element_text(family = "Times"),
        legend.position = "none")
```

Terraine Ruggedness slopes

Blue = African nations, Green = others



What proportion of these differences is below zero?

```
post %>%
  mutate(gamma_Africa      = b_rugged + `b_rugged:cont_africa`,
         gamma_notAfrica = b_rugged) %>%
  mutate(diff            = gamma_Africa - gamma_notAfrica) %>%
  summarise(Proportion_of_the_difference_below_0 = sum(diff < 0) / length(diff))

##   Proportion_of_the_difference_below_0
## 1                         0.00325
```

7.2 Symmetry of the linear interaction.

Consider for example the GDP and terrain ruggedness problem. The interaction there has two equally valid phrasings.

1. How much does the influence of ruggedness (on GDP) depend upon whether the nation is in Africa?
2. How much does the influence of being in Africa (on GDP) depend upon ruggedness?

While these two possibilities sound different to most humans, your golem thinks they are identical. (p. 223)

7.2.1 Buridan's interaction.

Recall the original equation.

$$\begin{aligned} \log_{\text{gdp}}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \gamma_i \text{rugged}_i + \beta_2 \text{cont_africa}_i \\ \gamma_i &= \beta_1 + \beta_3 \text{cont_africa}_i \end{aligned}$$

Next McElreath replaced γ_i with the expression for μ_i .

$$\begin{aligned} \mu_i &= \alpha + (\beta_1 + \beta_3 \text{cont_africa}_i) \times \text{rugged}_i + \beta_2 \text{cont_africa}_i \\ &= \alpha + \beta_1 \text{rugged}_i + \beta_3 \text{rugged}_i \times \text{cont_africa}_i + \beta_2 \text{cont_africa}_i \end{aligned}$$

And now we'll factor together the terms containing cont_africa_i .

$$\mu_i = \alpha + \beta_1 \text{rugged}_i + \underbrace{(\beta_2 + \beta_3 \text{rugged}_i)}_G \times \text{cont_africa}_i$$

And just as in the text, our G term looks a lot like the original γ_i term.

7.2.2 Africa depends upon ruggedness.

Here is our version of McElreath's Figure 7.6.

```
# new predictor data for `fitted()`
nd <-
  tibble(rugged      = rep(range(dd$rugged), times = 2),
         cont_africa = rep(0:1, each = 2))

# `fitted()`
f <-
  fitted(b7.5, newdata = nd) %>%
  as_tibble() %>%
```

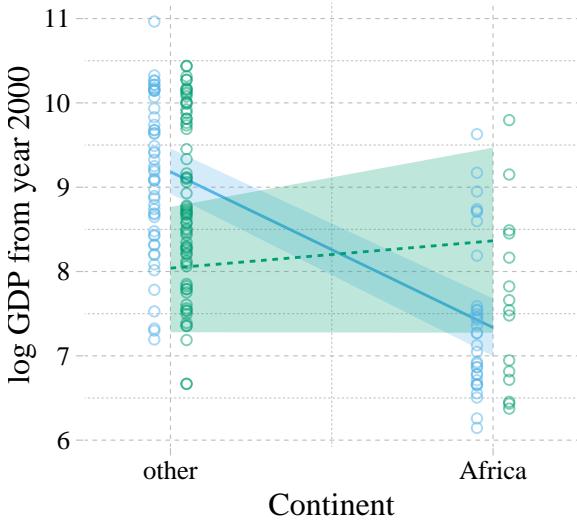
```

bind_cols(nd) %>%
  mutate(ox = rep(c(-0.05, 0.05), times = 2))

# augment the `dd` data a bit
dd %>%
  mutate(ox      = ifelse(rugged > median(rugged), 0.05, -0.05),
        cont_africa = cont_africa + ox) %>%
  select(cont_africa, everything()) %>%

# plot
ggplot(aes(x = cont_africa, color = factor(ox))) +
  geom_smooth(data = f,
              aes(y = Estimate, ymin = Q2.5, ymax = Q97.5,
                  fill = factor(ox), linetype = factor(ox)),
              stat = "identity",
              alpha = 1/4, size = 1/2) +
  geom_point(aes(y = log_gdp),
              alpha = 1/2, shape = 1) +
  scale_colour_pander() +
  scale_fill_pander() +
  scale_x_continuous("Continent", breaks = 0:1,
                     labels = c("other", "Africa")) +
  coord_cartesian(xlim = c(-.2, 1.2)) +
  ylab("log GDP from year 2000") +
  theme_pander() +
  theme(text = element_text(family = "Times")),
  legend.position = "none")

```



7.3 Continuous interactions

Though continuous interactions can be more challenging to interpret, they're just as easy to fit as interactions including dummies.

7.3.1 The data.

Look at the tulips.

```

library(rethinking)
data(tulips)

```

```
d <- tulips
str(d)

## 'data.frame': 27 obs. of 4 variables:
## $ bed   : Factor w/ 3 levels "a","b","c": 1 1 1 1 1 1 1 1 1 2 ...
## $ water : int 1 1 1 2 2 2 3 3 3 1 ...
## $ shade : int 1 2 3 1 2 3 1 2 3 1 ...
## $ blooms: num 0 0 111 183.5 59.2 ...
```

7.3.2 The un-centered models.

The likelihoods for the next two models are

$$\begin{aligned} \text{blooms}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta_1 \text{water}_i + \beta_2 \text{shade}_i \\ \alpha &\sim \text{Normal}(0, 100) \\ \beta_1 &\sim \text{Normal}(0, 100) \\ \beta_2 &\sim \text{Normal}(0, 100) \\ \sigma &\sim \text{Uniform}(0, 100) \end{aligned}$$

and

$$\begin{aligned} \text{blooms}_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta_1 \text{water}_i + \beta_2 \text{shade}_i + \beta_3 \text{water}_i \times \text{shade}_i \\ \alpha &\sim \text{Normal}(0, 100) \\ \beta_1 &\sim \text{Normal}(0, 100) \\ \beta_2 &\sim \text{Normal}(0, 100) \\ \beta_3 &\sim \text{Normal}(0, 100) \\ \sigma &\sim \text{Uniform}(0, 100) \end{aligned}$$

Load brms.

```
detach(package:rethinking, unload = T)
library(brms)
rm(tulips)
```

Here we continue with McElreath's very-flat priors for the multivariable and interaction models.

```
b7.6 <-
  brm(data = d, family = gaussian,
       blooms ~ 1 + water + shade,
       prior = c(prior(normal(0, 100), class = Intercept),
                 prior(normal(0, 100), class = b),
                 prior(uniform(0, 100), class = sigma)),
       iter = 2000, warmup = 1000, cores = 4, chains = 4,
       seed = 7)
```

```
## Warning: There were 58 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help. See
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

```
b7.7 <-
  update(b7.6,
    formula = blooms ~ 1 + water + shade + water:shade)

## Warning: There were 4 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help. See
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup

## Warning: Examine the pairs() plot to diagnose sampling problems
```

Much like in the text, these models yielded divergent transitions. Here, we'll try to combat them by following Stan's advice and “[increase] adapt_delta above 0.8.” While we're at it, we'll put better priors on σ .

```
b7.6 <-
  update(b7.6,
    prior = c(prior(normal(0, 100), class = Intercept),
              prior(normal(0, 100), class = b),
              prior(cauchy(0, 10), class = sigma)),
    control = list(adapt_delta = 0.9),
    seed = 7)

b7.7 <-
  update(b7.6,
    formula = blooms ~ 1 + water + shade + water:shade)
```

Increasing `adapt_delta` did the trick. Instead of `coeftab()`, we can also use `posterior_summary()`, which gets us most of the way there.

```
posterior_summary(b7.6) %>% round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	60.54	42.30	-21.72	145.01
## b_water	73.79	14.60	44.59	102.07
## b_shade	-40.56	14.66	-69.79	-11.18
## sigma	61.61	9.38	46.70	82.54
## lp__	-169.81	1.54	-173.72	-167.87

```
posterior_summary(b7.7) %>% round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	-103.45	62.11	-224.10	22.21
## b_water	158.20	28.69	98.59	211.95
## b_shade	42.30	28.46	-14.91	96.56
## b_water:shade	-42.43	13.28	-67.74	-15.43
## sigma	49.80	7.65	37.67	67.10
## lp__	-170.59	1.72	-174.81	-168.32

This is an example where HMC yielded point estimates notably different from MAP. However, look at the size of those posterior standard deviations (i.e., ‘Est.Error’ column)! The MAP estimates are well within a fraction of those *SDs*.

Anyway, let's look at WAIC.

```
b7.6 <- add_criterion(b7.6, "waic")
b7.7 <- add_criterion(b7.7, "waic")

w <- loo_compare(b7.6, b7.7, criterion = "waic")

print(w, simplify = F)
```

```
##      elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic    se_waic
## b7.7     0.0      0.0   -146.8      4.0      4.7     1.2    293.7     8.0
## b7.6    -5.3      2.7   -152.1      3.8      4.1     1.1    304.2     7.7
```

Here we use our `cbind()` trick to convert the difference from the elpd metric to the more traditional WAIC metric.

```
cbind(waic_diff = w[, 1] * -2,
      se        = w[, 2] * 2)
```

```
##      waic_diff      se
## b7.7    0.00000 0.000000
## b7.6   10.52269 5.367824
```

Why not compute the WAIC weights?

```
model_weights(b7.6, b7.7, weights = "waic")
```

```
##      b7.6      b7.7
## 0.005161551 0.994838449
```

As in the text, almost all the weight went to the interaction model, `b7.7`.

7.3.3 Center and re-estimate.

To *center* a variable means to create a new variable that contains the same information as the original, but has a new mean of zero. For example, to make centered versions of `shade` and `water`, just subtract the mean of the original from each value. (p. 230, *emphasis* in the original)

Here's a tidyverse way to center the predictors.

```
d <-
d %>%
  mutate(shade_c = shade - mean(shade),
        water_c = water - mean(water))
```

Now refit the models with our shiny new centered predictors.

```
b7.8 <-
  brm(data = d, family = gaussian,
       blooms ~ 1 + water_c + shade_c,
       prior = c(prior(normal(130, 100), class = Intercept),
                  prior(normal(0, 100), class = b),
                  prior(cauchy(0, 10), class = sigma)),
       iter = 2000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = 0.9),
       seed = 7)

b7.9 <-
  update(b7.8,
         formula = blooms ~ 1 + water_c + shade_c + water_c:shade_c)
```

Check out the results.

```
posterior_summary(b7.8) %>% round(digits = 2)
```

```
##             Estimate Est.Error   Q2.5   Q97.5
## b_Intercept    128.97     11.58 105.40 151.69
## b_water_c      74.31      14.27  46.11 102.59
## b_shade_c     -40.73     14.52 -69.06 -12.14
## sigma          61.42      9.14   46.34  82.07
## lp__         -168.93     1.49 -172.68 -167.03
```

```
posterior_summary(b7.9) %>% round(digits = 2)
```

```
##             Estimate Est.Error   Q2.5   Q97.5
## b_Intercept    129.07      9.66 110.30 148.03
## b_water_c      74.60     11.31  51.68  96.81
## b_shade_c     -40.96     11.90 -63.89 -17.48
## b_water_c:shade_c -51.62     14.02 -77.80 -23.55
## sigma          49.55      7.28   37.93  67.02
## lp__         -168.53     1.67 -172.56 -166.30
```

And okay fine, if you really want a `coeftab()`-like summary, here's a way to do it.

```
tibble(model = str_c("b7.", 8:9)) %>%
  mutate(fit = purrr::map(model, get)) %>%
  mutate(tidy = purrr::map(fit, broom::tidy)) %>%
  unnest(tidy) %>%
  filter(term != "lp__") %>%
  select(term, estimate, model) %>%
  spread(key = model, value = estimate) %>%
  mutate_if(is.double, round, digits = 2)
```

```
## # A tibble: 5 x 3
##   term           b7.8   b7.9
##   <chr>        <dbl>  <dbl>
## 1 b_Intercept   129.    129.
## 2 b_shade_c     -40.7   -41.0
## 3 b_water_c      74.3    74.6
## 4 b_water_c:shade_c NA     -51.6
## 5 sigma          61.4    49.6
```

Anyway, centering helped a lot. Now, not only do the results in the text match up better than those from Stan, but the ‘Est.Error’ values are uniformly smaller.

7.3.3.1 Estimation worked better.

Nothing to add, here.

7.3.3.2 Estimates changed less across models.

On page 231, we read:

The interaction parameter always factors into generating a prediction. Consider for example a tulip at the average moisture and shade levels, 2 in each case. The expected blooms for such a tulip is:

$$\mu_i | \text{shade}_{i=2}, \text{water}_{i=2} = \alpha + \beta_{\text{water}}(2) + \beta_{\text{shade}}(2) + \beta_{\text{water} \times \text{shade}}(2 \times 2)$$

So to figure out the effect of increasing water by 1 unit, you have to use all of the β parameters. Plugging in the [HMC] values for the un-centered interaction model, [b7.7], we get:

$$\mu_i | \text{shade}_{i=2}, \text{water}_{i=2} = -107.1 + 159.9(2) + 44.0(2) - 43.2(2 \times 2)$$

With our brms workflow, we use `fixef()` to compute the predictions.

```
k <- fixef(b7.7)
k[1] + k[2] * 2 + k[3] * 2 + k[4] * 2 * 2

## [1] 128.0883
```

Even though our HMC parameters differ a bit from the MAP estimates McElreath reported in the text, the value they predicted matches quite closely with the one in the text. Same thing for the next one.

```
k <- fixef(b7.9)
k[1] + k[2] * 0 + k[3] * 0 + k[4] * 0 * 0

## [1] 129.0661
```

Here are the coefficient summaries for the centered model.

```
print(b7.9)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: blooms ~ water_c + shade_c + water_c:shade_c
##   Data: d (Number of observations: 27)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     129.07     9.66   110.30   148.03      4457 1.00
## water_c        74.60    11.31    51.68    96.81      4878 1.00
## shade_c       -40.96    11.90   -63.89   -17.48      5516 1.00
## water_c:shade_c -51.62    14.02   -77.80   -23.55      4515 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma     49.55     7.28    37.93    67.02      3729 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

7.3.4 Plotting implied predictions.

Now we're ready for the bottom row of Figure 7.7. Here's our variation on McElreath's tryptych loop code, adjusted for brms and ggplot2.

```
# loop over values of `water_c` and plot predictions
shade_seq <- -1:1

for(w in -1:1){
  # define the subset of the original data
  dt <- d[d$water_c == w, ]
  # defining our new data
  nd <- tibble(water_c = w, shade_c = shade_seq)
```

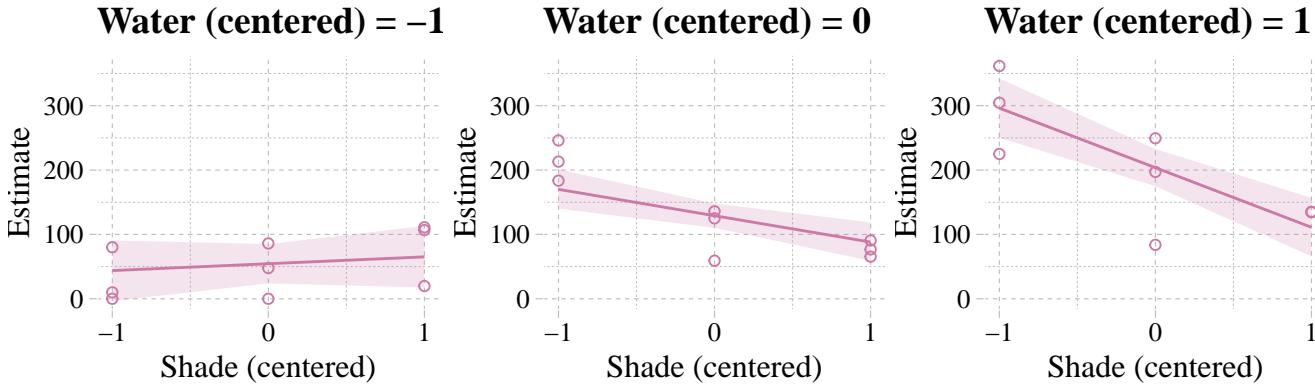
```

# use our sampling skills, like before
f <-
  fitted(b7.9, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

# specify our custom plot
fig <-
  ggplot() +
  geom_smooth(data = f,
    aes(x = shade_c, y = Estimate, ymin = Q2.5, ymax = Q97.5),
    stat = "identity",
    fill = "#CC79A7", color = "#CC79A7", alpha = 1/5, size = 1/2) +
  geom_point(data = dt,
    aes(x = shade_c, y = blooms),
    shape = 1, color = "#CC79A7") +
  coord_cartesian(xlim = range(d$shade_c),
    ylim = range(d$blooms)) +
  scale_x_continuous("Shade (centered)", breaks = c(-1, 0, 1)) +
  labs("Blooms",
    title = paste("Water (centered) =", w)) +
  theme_pander() +
  theme(text = element_text(family = "Times"))

# plot that joint
plot(fig)
}

```



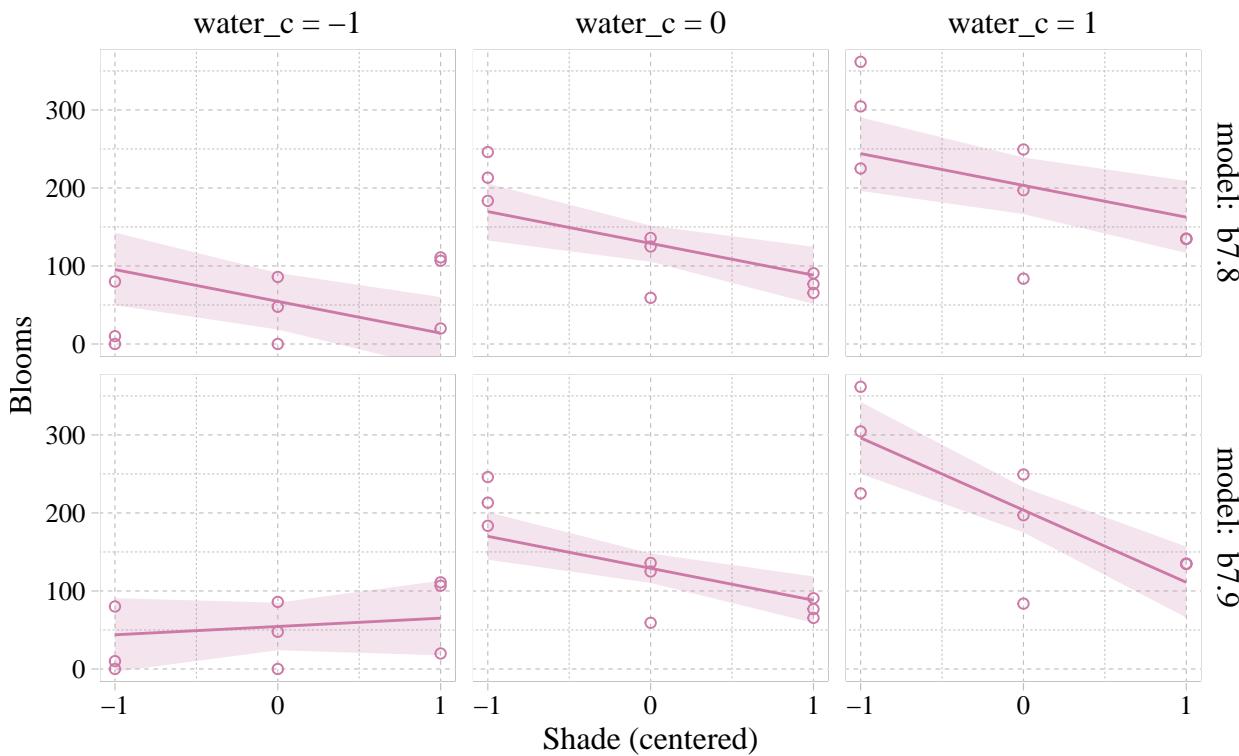
But we don't necessarily need a loop. We can achieve all of McElreath's Figure 7.7 with `fitted()`, some data wrangling, and a little help from `ggplot2::facet_grid()`.

```

# `fitted()` for model b7.8
fitted(b7.8) %>%
  as_tibble() %>%
# add `fitted()` for model b7.9
  bind_rows(
    fitted(b7.9) %>%
      as_tibble())
  ) %>%
# we'll want to index the models
  mutate(fit = rep(c("b7.8", "b7.9"), each = 27)) %>%
# here we add the data, `d`
  bind_cols(bind_rows(d, d)) %>%
# these will come in handy for `ggplot2::facet_grid()`
  mutate(x_grid = paste("water_c =", water_c),
    y_grid = paste("model: ", fit)) %>%

```

```
# plot!
ggplot(aes(x = shade_c)) +
  geom_smooth(aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = "#CC79A7", color = "#CC79A7", alpha = 1/5, size = 1/2) +
  geom_point(aes(y = blooms, group = x_grid),
             shape = 1, color = "#CC79A7") +
  coord_cartesian(xlim = range(d$shade_c),
                  ylim = range(d$blooms)) +
  scale_x_continuous("Shade (centered)", breaks = c(-1, 0, 1)) +
  ylab("Blooms") +
  theme_pander() +
  theme(text = element_text(family = "Times"),
        panel.background = element_rect(color = "black")) +
  facet_grid(y_grid ~ x_grid)
```



7.4 Interactions in design formulas

The brms syntax generally follows the design formulas typical of `lm()`. Hopefully this is all old hat.

7.5 Summary Bonus: `marginal_effects()`

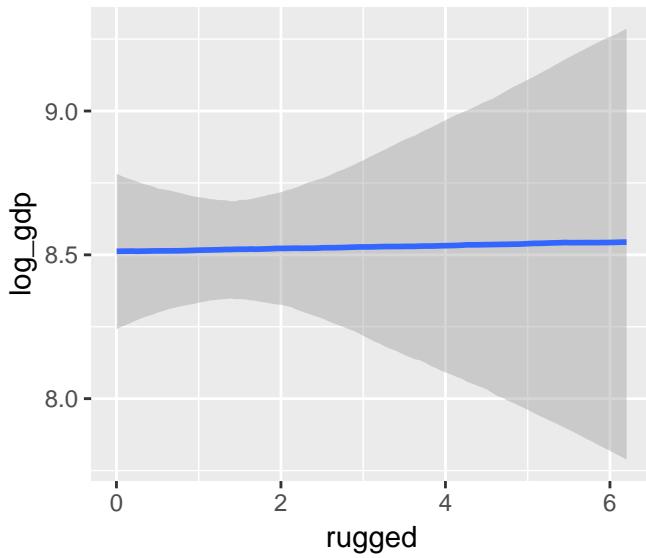
The brms package includes the `marginal_effects()` function as a convenient way to look at simple effects and two-way interactions. Recall the simple univariable model, b7.3:

```
b7.3$formula
```

```
## log_gdp ~ 1 + rugged
```

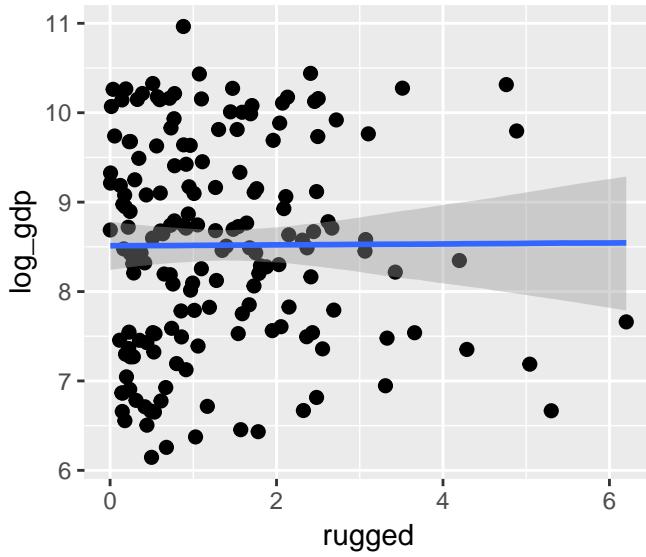
We can look at the regression line and its percentile-based intervals like so:

```
marginal_effects(b7.3)
```



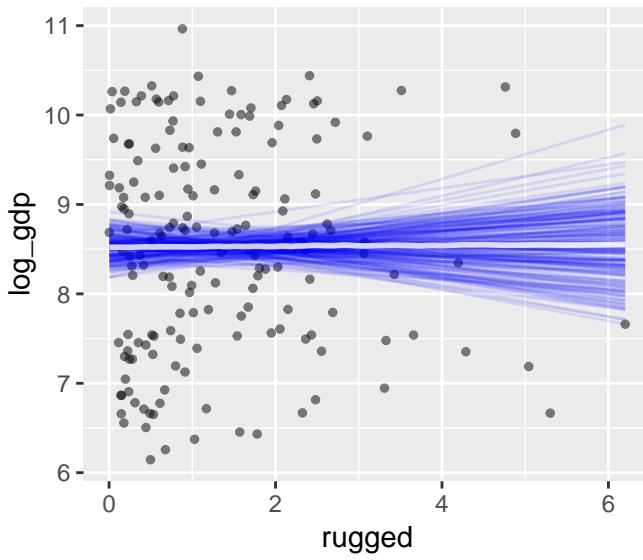
If we nest `marginal_effects()` within `plot()` with a `points = T` argument, we can add the original data to the figure.

```
plot(marginal_effects(b7.3), points = T)
```



We can further customize the plot. For example, we can replace the intervals with a spaghetti plot. While we're at it, we can use `point_args` to adjust the `geom_jitter()` parameters.

```
plot(marginal_effects(b7.3,
                      spaghetti = T, nsamples = 200),
      points = T,
      point_args = c(alpha = 1/2, size = 1))
```

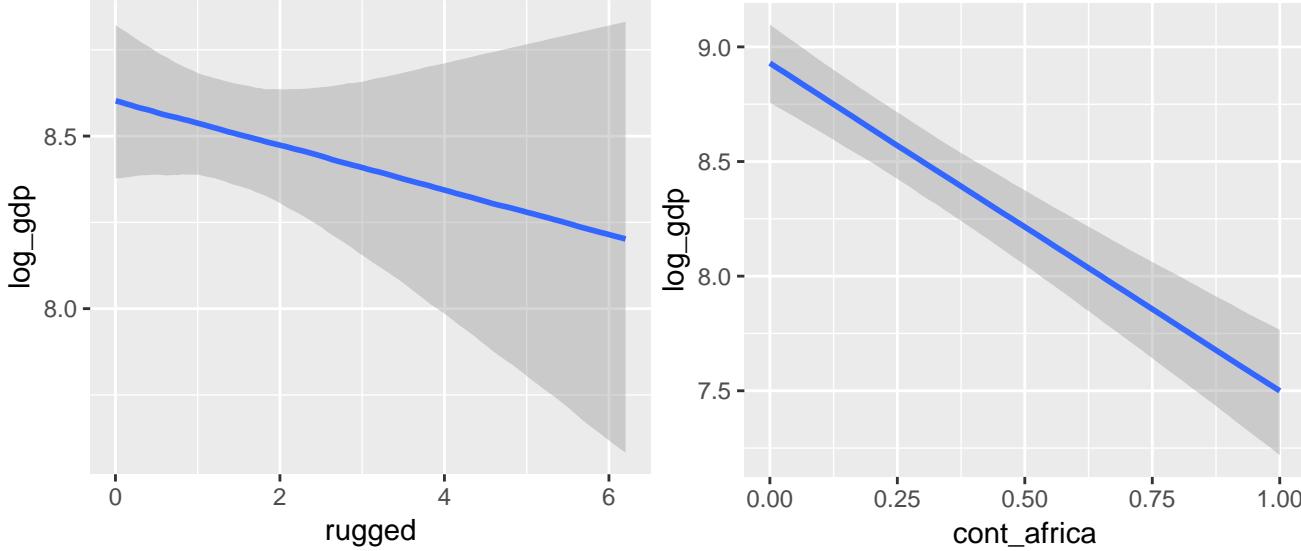


With multiple predictors, things get more complicated. Consider our multivariable, non-interaction model, b7.4.

```
b7.4$formula
```

```
## log_gdp ~ rugged + cont_africa
```

```
marginal_effects(b7.4)
```



We got one plot for each predictor, controlling the other predictor at zero. Note how the plot for cont_africa treated it as a continuous variable. This is because the variable was saved as an integer in the original data set:

```
b7.4$data %>%
  glimpse()
```

```
## Observations: 170
## Variables: 3
## $ log_gdp      <dbl> 7.492609, 8.216929, 9.933263, 9.407032, 7.792343, 9.212541, 10.143191, 10.2...
## $ rugged       <dbl> 0.858, 3.427, 0.769, 0.775, 2.688, 0.006, 0.143, 3.513, 1.672, 1.780, 0.388...
## $ cont_africa <int> 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
```

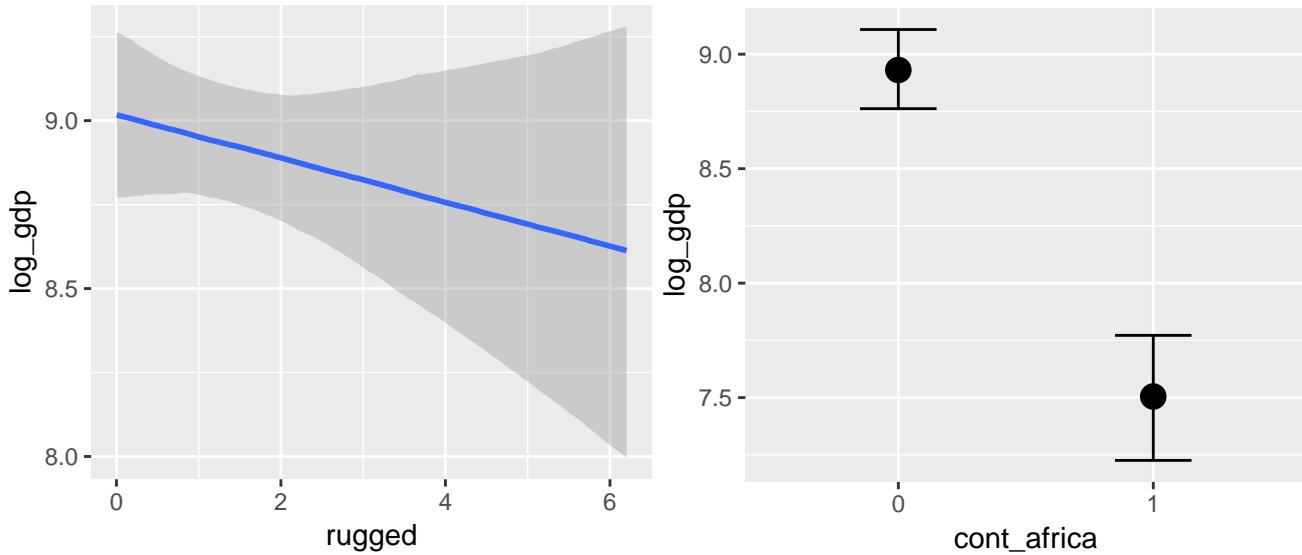
One way to fix that is to adjust the data set and refit the model.

```
d_factor <-
  b7.4$data %>%
  mutate(cont_africa = factor(cont_africa))

b7.4_factor <- update(b7.4, newdata = d_factor)
```

Using the `update()` syntax often speeds up the re-fitting process.

```
marginal_effects(b7.4_factor)
```



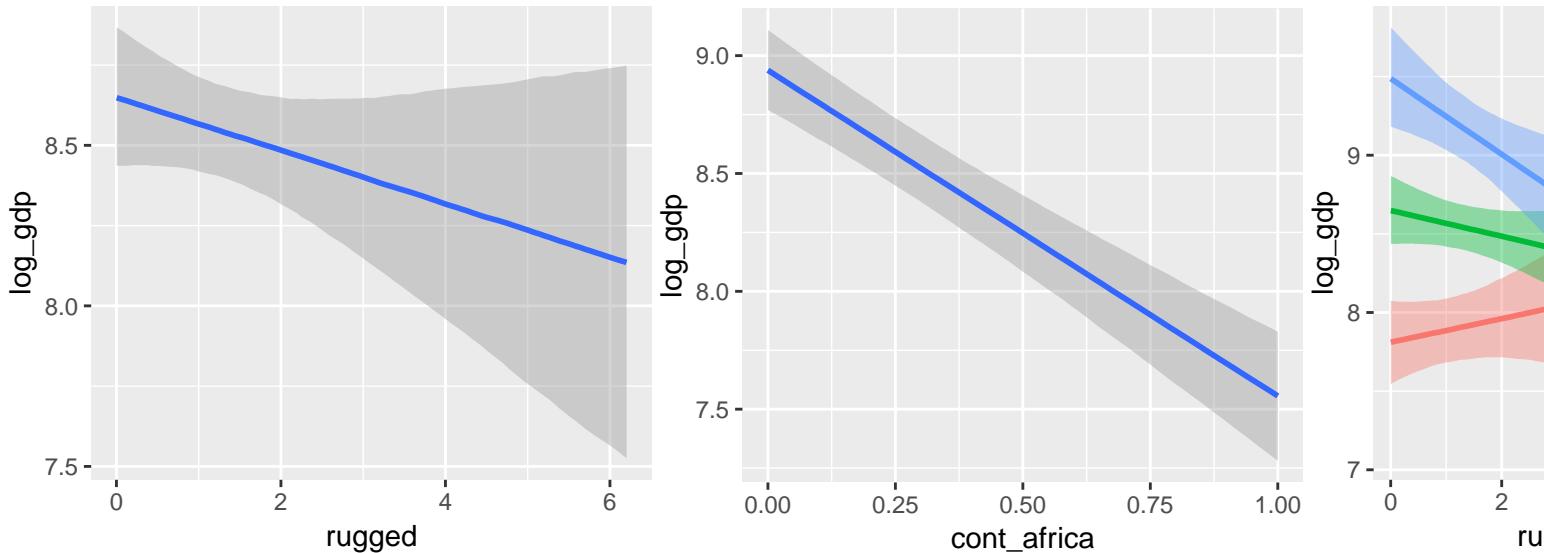
Now our second marginal plot more clearly expresses the `cont_africa` predictor as categorical.

Things get more complicated with the interaction model, `b7.5`.

```
b7.5$formula
```

```
## log_gdp ~ rugged + cont_africa + rugged:cont_africa
```

```
marginal_effects(b7.5)
```



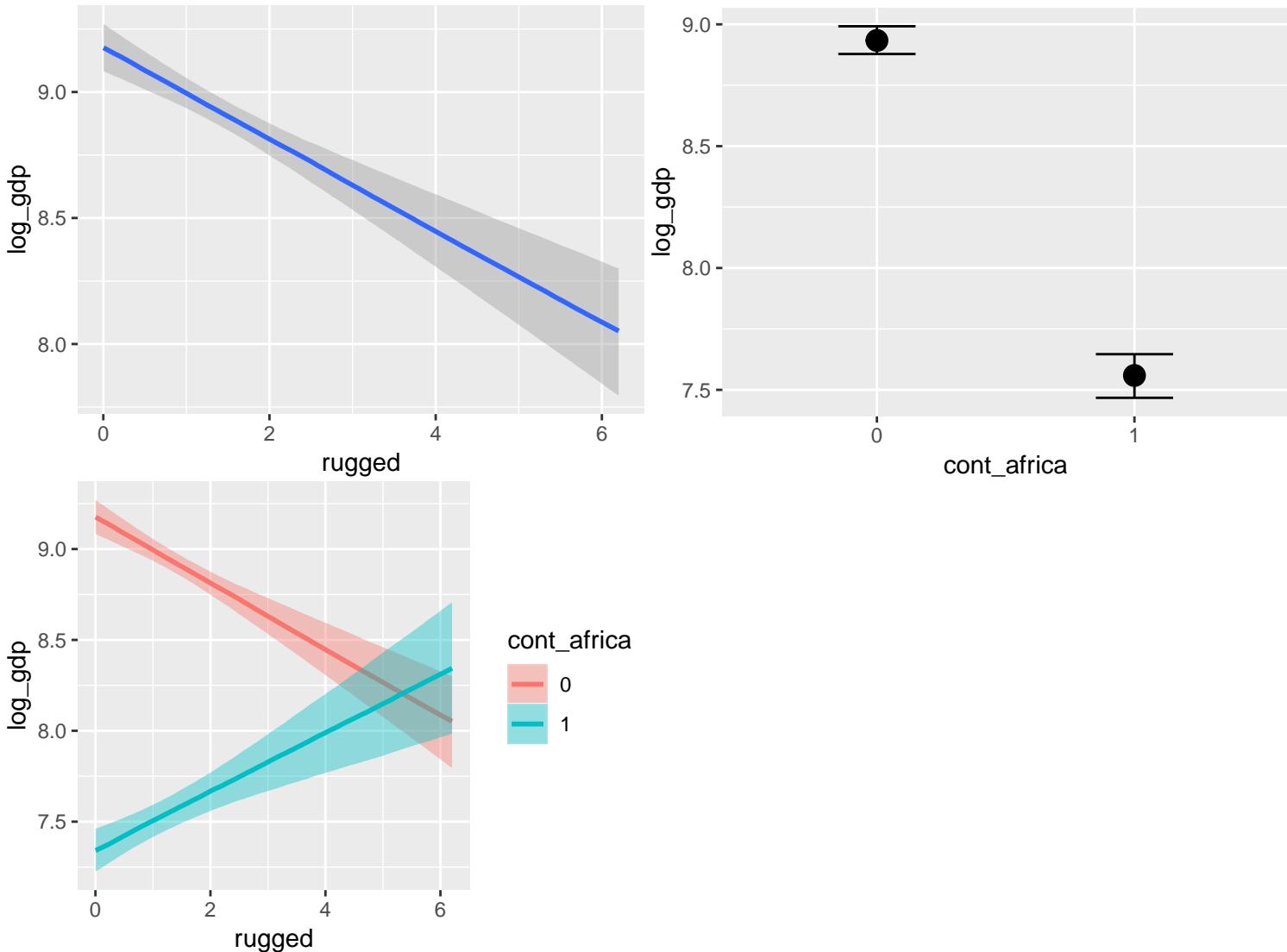
The `marginal_effects()` function defaults to expressing interactions such that the first variable in the term—in this case, `rugged`—is on the x axis and the second variable in the term—`cont_africa`, treated as an integer—is depicted in three lines corresponding its mean and its mean +/- one standard deviation. This is great for continuous variables, but incoherent for categorical ones. The fix is, you guessed it, to refit the model after adjusting the data.

```
d_factor <-
  b7.5$data %>%
  mutate(cont_africa = factor(cont_africa))

b7.5_factor <- update(b7.5, newdata = d_factor)
```

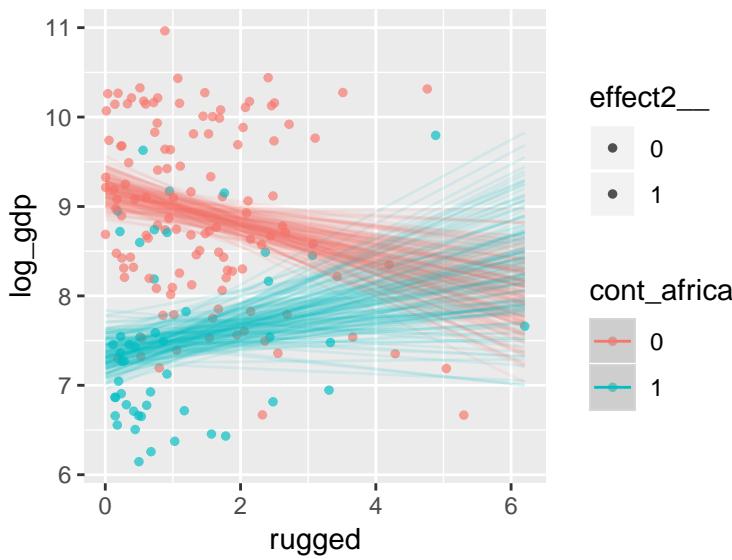
Just for kicks, we'll use `probs = c(.25, .75)` to return [50% intervals](#), rather than the conventional 95%.

```
marginal_effects(b7.5_factor,
                  probs = c(.25, .75))
```



With the `effects` argument, we can just return the interaction effect, which is where all the action's at. While we're at it, we'll use `plot()` to change some of the settings.

```
plot(marginal_effects(b7.5_factor,
                      effects = "rugged:cont_africa",
                      spaghetti = T, nsamples = 150),
     points = T,
     point_args = c(alpha = 2/3, size = 1), mean = F)
```



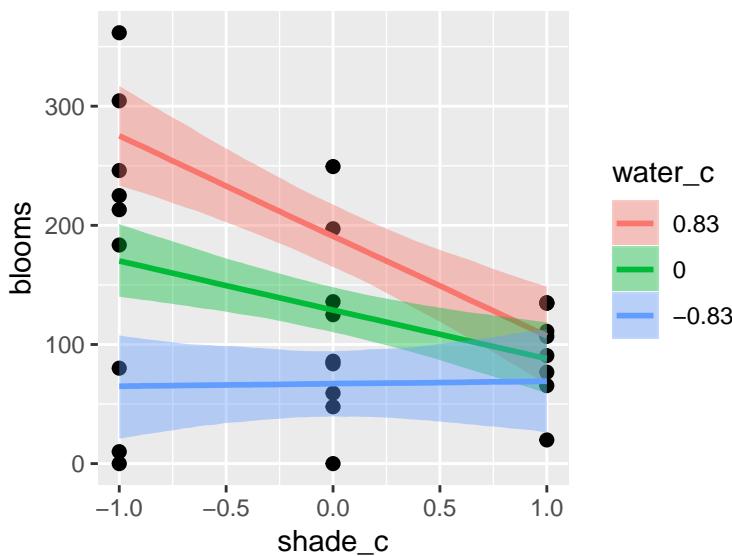
Note, the ordering of the variables matters for the interaction term. Consider our interaction model for the tulips data.

```
b7.9$formula
```

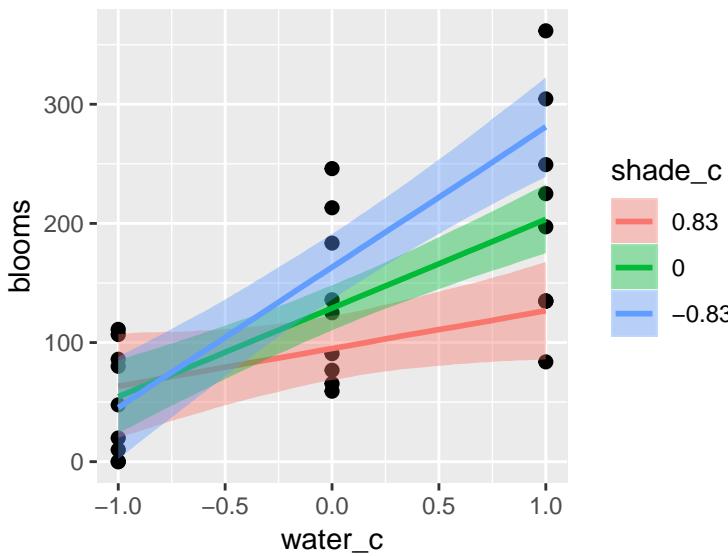
```
## blooms ~ water_c + shade_c + water_c:shade_c
```

The plot tells a slightly different story, depending on whether you specify `effects = "shade_c:water_c"` or `effects = "water_c:shade_c"`.

```
plot(marginal_effects(b7.9,
                      effects = "shade_c:water_c"),
      points = T)
```

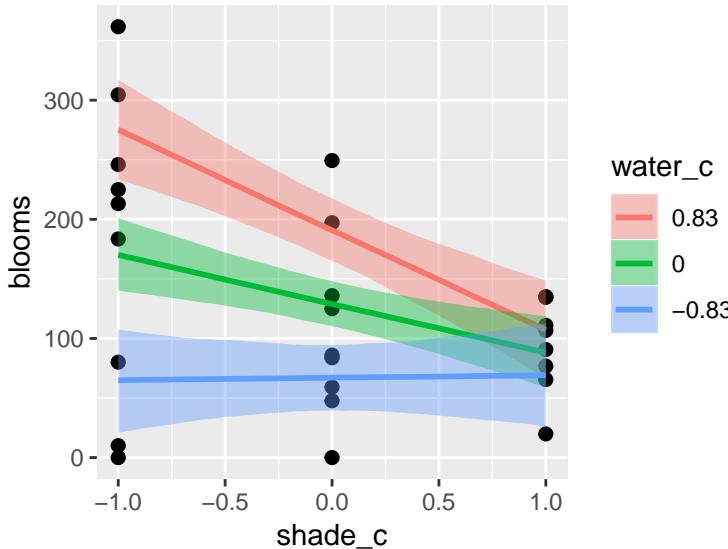


```
plot(marginal_effects(b7.9,
                      effects = "water_c:shade_c"),
      points = T)
```



One might want to evaluate the effects of the second term in the interaction—water_c, in this case—at values other than the mean and the mean \pm one standard deviation. When we reproduced the bottom row of Figure 7.7, we expressed the interaction based on values -1, 0, and 1 for water_c. We can do that, here, by using the int_conditions argument. It expects a list, so we'll put our desired water_c values in just that.

```
ic <-  
  list(water.c = c(-1, 0, 1))  
  
plot(marginal_effects(b7.9,  
                      effects = "shade_c:water_c",  
                      int_conditions = ic),  
      points = T)
```



Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods  base
##
## other attached packages:
## [1] ggthemes_4.0.1     forcats_0.3.0     stringr_1.4.0      dplyr_0.8.0.1
## [5] purrr_0.2.5        readr_1.1.1      tidyverse_1.2.1    tibble_2.1.1
## [9] tidyverse_1.2.1    brms_2.8.8       Rcpp_1.0.1        rstan_2.18.2
## [13] StanHeaders_2.18.0-1 ggplot2_3.1.1
##
## loaded via a namespace (and not attached):
## [1] nlme_3.1-137      matrixStats_0.54.0   xts_0.10-2        lubridate_1.7.4
## [5] threejs_0.3.1     httr_1.3.1         rprojroot_1.3-2   tools_3.5.1
## [9] backports_1.1.4    utf8_1.1.4         R6_2.3.0          DT_0.4
## [13] lazyeval_0.2.2    colorspace_1.3-2   withr_2.1.2       tidyselect_0.2.5
## [17] gridExtra_2.3     prettyunits_1.0.2   processx_3.2.1   Broddingnag_1.2-6
## [21] compiler_3.5.1    rvest_0.3.2        cli_1.0.1         xml2_1.2.0
## [25] shinyjs_1.0       labeling_0.3       colourpicker_1.0  bookdown_0.9
## [29] scales_1.0.0      dygraphs_1.1.1.5   mvtnorm_1.0-10   ggridges_0.5.0
## [33] callr_3.1.0      digest_0.6.18     rmarkdown_1.10    base64enc_0.1-3
## [37] pkgconfig_2.0.2   htmltools_0.3.6   readxl_1.1.0     htmlwidgets_1.2
## [41] rlang_0.3.4       rstudioapi_0.7    shiny_1.1.0       generics_0.0.2
## [45] zoo_1.8-2         jsonlite_1.5      crosstalk_1.0.0  gtools_3.8.1
## [49] inline_0.3.15    magrittr_1.5      loo_2.1.0         bayesplot_1.6.0
## [53] Matrix_1.2-14    fansi_0.4.0       munsell_0.5.0    abind_1.4-5
## [57] stringi_1.4.3    yaml_2.1.19      MASS_7.3-50       pkgbuild_1.0.2
## [61] plyr_1.8.4        grid_3.5.1        promises_1.0.1   crayon_1.3.4
## [65] miniUI_0.1.1.1   lattice_0.20-35  haven_1.1.2      pander_0.6.2
## [69] hms_0.4.2         knitr_1.20       ps_1.2.1         pillar_1.3.1
## [73] igraph_1.2.1     markdown_0.8      shinystan_2.5.0  codetools_0.2-15
## [77] reshape2_1.4.3   stats4_3.5.1     rstantools_1.5.1 glue_1.3.1.9000
## [81] evaluate_0.10.1  modelr_0.1.2     httpuv_1.4.4.2   cellranger_1.1.0
## [85] gttable_0.3.0    assertthat_0.2.0  xfun_0.3         mime_0.5
## [89] xtable_1.8-2     broom_0.5.1      coda_0.19-2      later_0.7.3
## [93] rsconnect_0.8.8  shinythemes_1.1.1 bridgesampling_0.6-0
```


Chapter 8

Markov Chain Monte Carlo

“This chapter introduces one of the more marvelous examples of how Fortuna and Minerva cooperate: the estimation of posterior probability distributions using a stochastic process known as Markov chain Monte Carlo (MCMC) estimation” (p. 241). Though we’ve been using MCMC via the brms package for chapters, now, this chapter should clarify some of the details.

8.1 Good King Markov and His island kingdom

In this version of the code, we’ve added `set.seed()`, which helps make the exact results reproducible.

```
set.seed(8)

num_weeks <- 1e5
positions <- rep(0, num_weeks)
current <- 10
for (i in 1:num_weeks) {
  # record current position
  positions[i] <- current
  # flip coin to generate proposal
  proposal <- current + sample(c(-1, 1), size = 1)
  # now make sure he loops around the archipelago
  if (proposal < 1) proposal <- 10
  if (proposal > 10) proposal <- 1
  # move?
  prob_move <- proposal / current
  current <- ifelse(runif(1) < prob_move, proposal, current)
}
```

In this chapter, we’ll borrow a theme, `theme_ipsum()`, from the `hrbrthemes` package.

```
# install.packages("hrbrthemes", dependencies = T)
library(hrbrthemes)
```

Figure 8.2.a.

```
library(tidyverse)

tibble(week    = 1:1e5,
      island = positions) %>%
  ggplot(aes(x = week, y = island)) +
  geom_point(shape = 1) +
```

```
scale_x_continuous(breaks = seq(from = 0, to = 100, by = 20)) +
scale_y_continuous(breaks = seq(from = 0, to = 10, by = 2)) +
coord_cartesian(xlim = 0:100,
                 ylim = 1:10) +
labs(title    = "Behold: The Metropolis algorithm in action!",
     subtitle = "The dots show the king's path over the first 100 weeks.") +
theme_ipsum()
```

Behold: The Metropolis algorithm in action!

The dots show the king's path over the first 100 weeks.

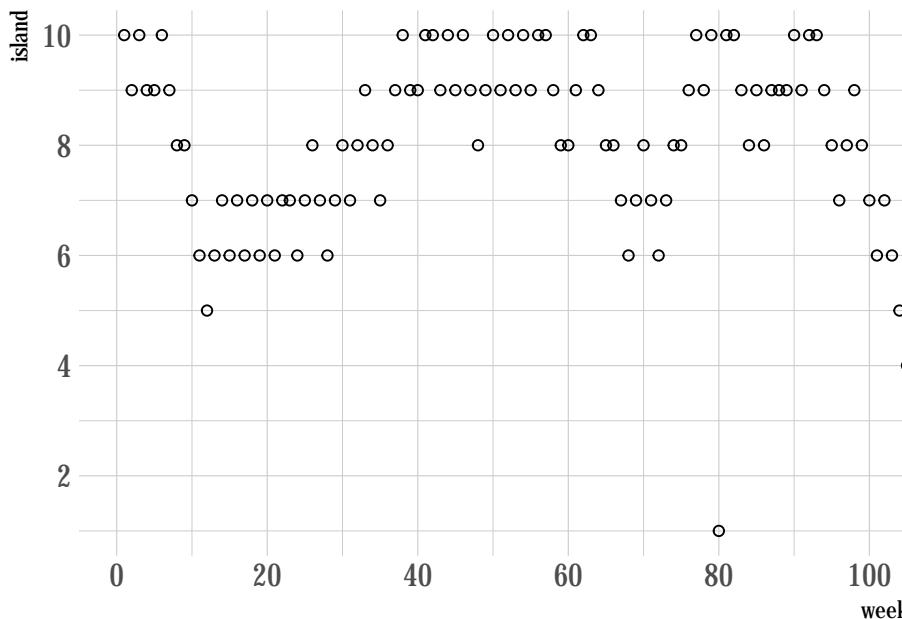


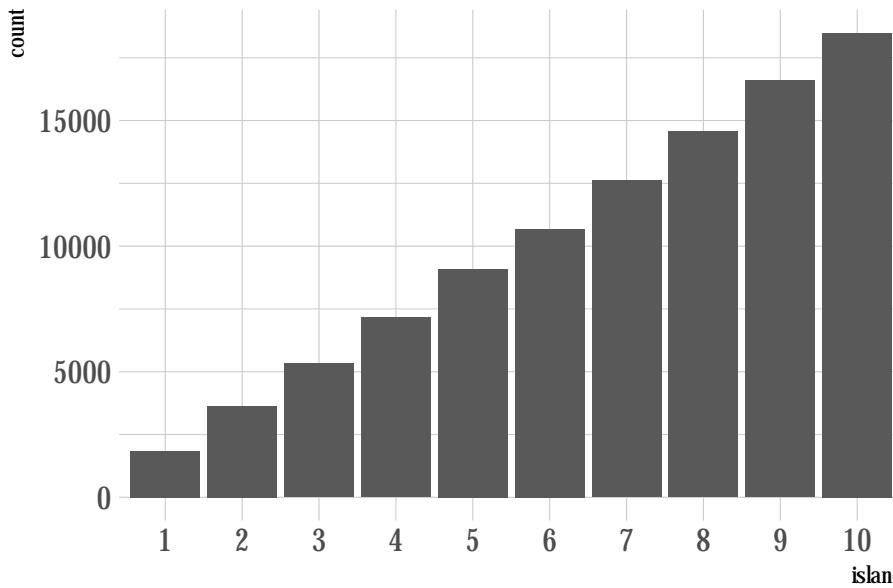
Figure 8.2.b.

```
tibble(week    = 1:1e5,
       island = positions) %>%
mutate(island = factor(island)) %>%

ggplot(aes(x = island)) +
geom_bar() +
labs(title    = "Old Metropolis shines in the long run.",
     subtitle = "Sure enough, the time the king spent on each island was\nproportional to its population size") +
theme_ipsum()
```

Old Metropolis shines in the long run.

Sure enough, the time the king spent on each island was proportional to its population size.



8.2 Markov chain Monte Carlo

“The metropolis algorithm is the grandparent of several different strategies for getting samples from unknown posterior distributions” (p. 245). If you’re interested, Robert and Casells wrote a [good historical overview of MCMC](#).

8.3 Easy HMC: `map2stan brm()`

Here we load the `rugged` data.

```
library(rethinking)
data(rugged)
d <- rugged
```

Switch from `rethinking` to `brms`.

```
detach(package:rethinking)
library(brms)
rm(rugged)
```

It takes just a sec to do a little data manipulation.

```
d <-
  d %>%
  mutate(log_gdp = log(rgdppc_2000))

dd <-
  d %>%
  drop_na(rgdppc_2000)
```

In the context of this chapter, it doesn’t make sense to translate McElreath’s `m8.1 map()` code to `brm()` code. Below, we’ll just go directly to the `brm()` variant of his `m8.1stan`.

8.3.1 Preparation.

When working with brms, you don't need to do the data processing McElreath did on pages 248 and 249. If you wanted to, however, here's how you might do it within the tidyverse.

```
dd.trim <-  
  dd %>%  
    select(log_gdp, rugged, cont_africa)  
  
str(dd.trim)
```

8.3.2 Estimation.

Finally, we get to work that sweet HMC.

```
b8.1 <-  
  brm(data = dd, family = gaussian,  
    log_gdp ~ 1 + rugged + cont_africa + rugged:cont_africa,  
    prior = c(prior(normal(0, 100), class = Intercept),  
              prior(normal(0, 10), class = b),  
              prior(cauchy(0, 2), class = sigma)),  
    seed = 8)
```

Now we have officially ditched the uniform distribution for σ . We'll only see it again in special cases for pedagogical purposes. Here's the posterior:

```
print(b8.1)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: log_gdp ~ 1 + rugged + cont_africa + rugged:cont_africa
## Data: dd (Number of observations: 170)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  9.22     0.14     8.94     9.50      2792 1.00
## rugged                   -0.20     0.08    -0.35    -0.05      2782 1.00
## cont_africa                -1.94     0.23    -2.39    -1.50      2900 1.00
## rugged:cont_africa         0.39     0.13     0.12     0.65      2741 1.00
##
## Family Specific Parameters:
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma                   0.95     0.05     0.86     1.06      4183 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Do note a couple things: If you look closely at the summary information at the top, you'll see that the `brms::brm()` function defaults to `chains = 4`. If you check the manual, you'll see it also defaults to `cores = 1`. You'll also note it defaults to `iter = 2000`, `warmup = 1000`. Also of note, McElreath's `rthinking::precis()` returns highest posterior density intervals (HPDIs) when summarizing `map2stan()` models. Not so with brms. If you want HPDIs, you'll have to use the convenience functions from the tidybayes package.

```

library(tidybayes)

post <- posterior_samples(b8.1)

post %>%
  gather() %>%
  group_by(key) %>%
  mean_hdi(value, .width = .89) # note our rare use of 89% intervals

## # A tibble: 6 x 7
##   key           value   .lower   .upper .width .point .interval
##   <chr>        <dbl>    <dbl>    <dbl>  <dbl> <chr>   <chr>
## 1 b_cont_africa -1.94   -2.30   -1.56   0.89 mean    hdi
## 2 b_Intercept     9.22    8.99    9.45   0.89 mean    hdi
## 3 b_rugged      -0.201   -0.330  -0.0809  0.89 mean    hdi
## 4 b_rugged:cont_africa 0.391    0.157    0.586  0.89 mean    hdi
## 5 lp__          -249.    -251.    -246.   0.89 mean    hdi
## 6 sigma         0.952    0.868    1.03   0.89 mean    hdi

```

8.3.3 Sampling again, in parallel.

Here we sample in parallel by adding `cores = 4`.

```

b8.1_4chains_4cores <-
  update(b8.1,
         cores = 4)

```

This model sampled so fast that it really didn't matter if we sampled in parallel or not. It will for others.

```

print(b8.1_4chains_4cores)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: log_gdp ~ 1 + rugged + cont_africa + rugged:cont_africa
##   Data: dd (Number of observations: 170)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  9.22     0.14    8.94    9.50     2953 1.00
## rugged                   -0.20     0.08   -0.35   -0.05     2629 1.00
## cont_africa                -1.94    0.23   -2.38   -1.49     2495 1.00
## rugged:cont_africa         0.39     0.13    0.13    0.64     2542 1.00
##
## Family Specific Parameters:
##                           Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma            0.95      0.05     0.85    1.06     4595 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

8.3.4 Visualization.

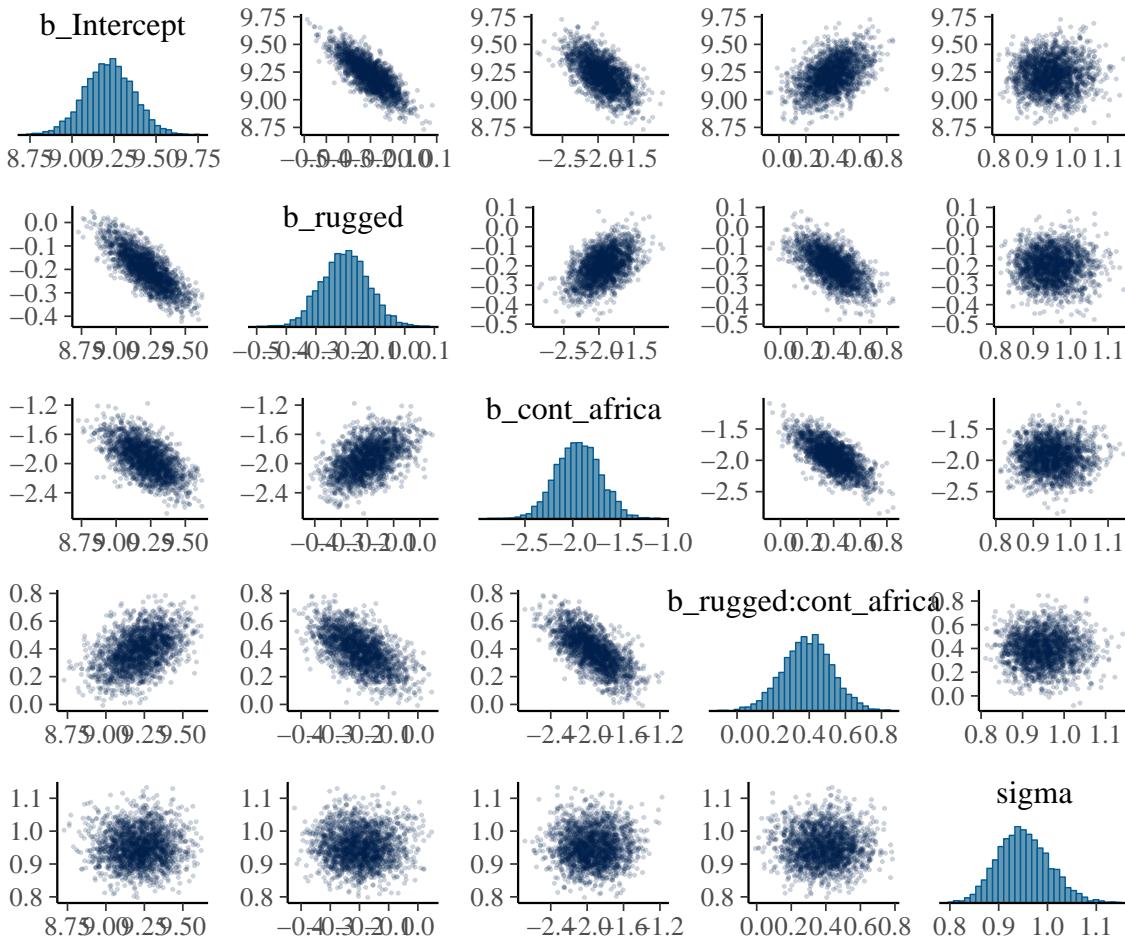
Unlike the way rethinking's `extract.samples()` yields a list, brms's `posterior_samples()` returns a data frame.

```
post <- posterior_samples(b8.1)
str(post)

## 'data.frame': 4000 obs. of 6 variables:
## $ b_Intercept      : num 9.48 9.19 9.2 9.2 9.39 ...
## $ b_rugged         : num -0.349 -0.197 -0.143 -0.167 -0.305 ...
## $ b_cont_africa    : num -2.52 -1.74 -1.92 -1.92 -2.2 ...
## $ b_rugged:cont_africa: num 0.719 0.287 0.421 0.328 0.524 ...
## $ sigma             : num 0.941 0.949 0.964 0.934 0.972 ...
## $ lp__              : num -250 -247 -247 -246 -247 ...
```

As with McElreath's rethinking, brms allows users to put the post data frame or the brmsfit object directly in pairs().

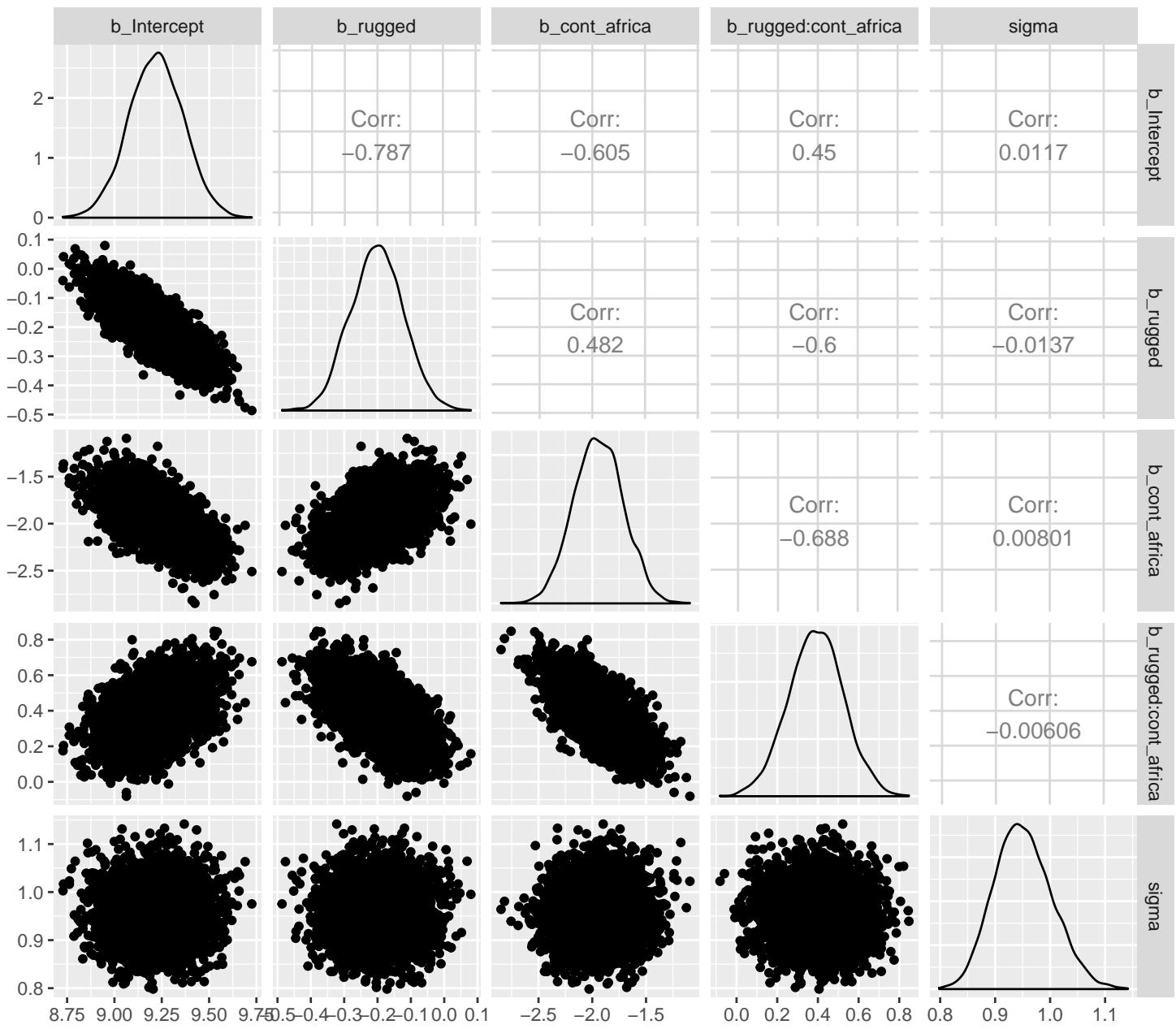
```
pairs(b8.1,
      off_diag_args = list(size = 1/5, alpha = 1/5))
```



Another nice way to customize your pairs plot is with the [GGally package](#).

```
library(GGally)
```

```
post %>%
  select(b_Intercept:sigma) %>%
  ggpairs()
```



Since GGally returns a ggplot2 object, you can customize it as you please.

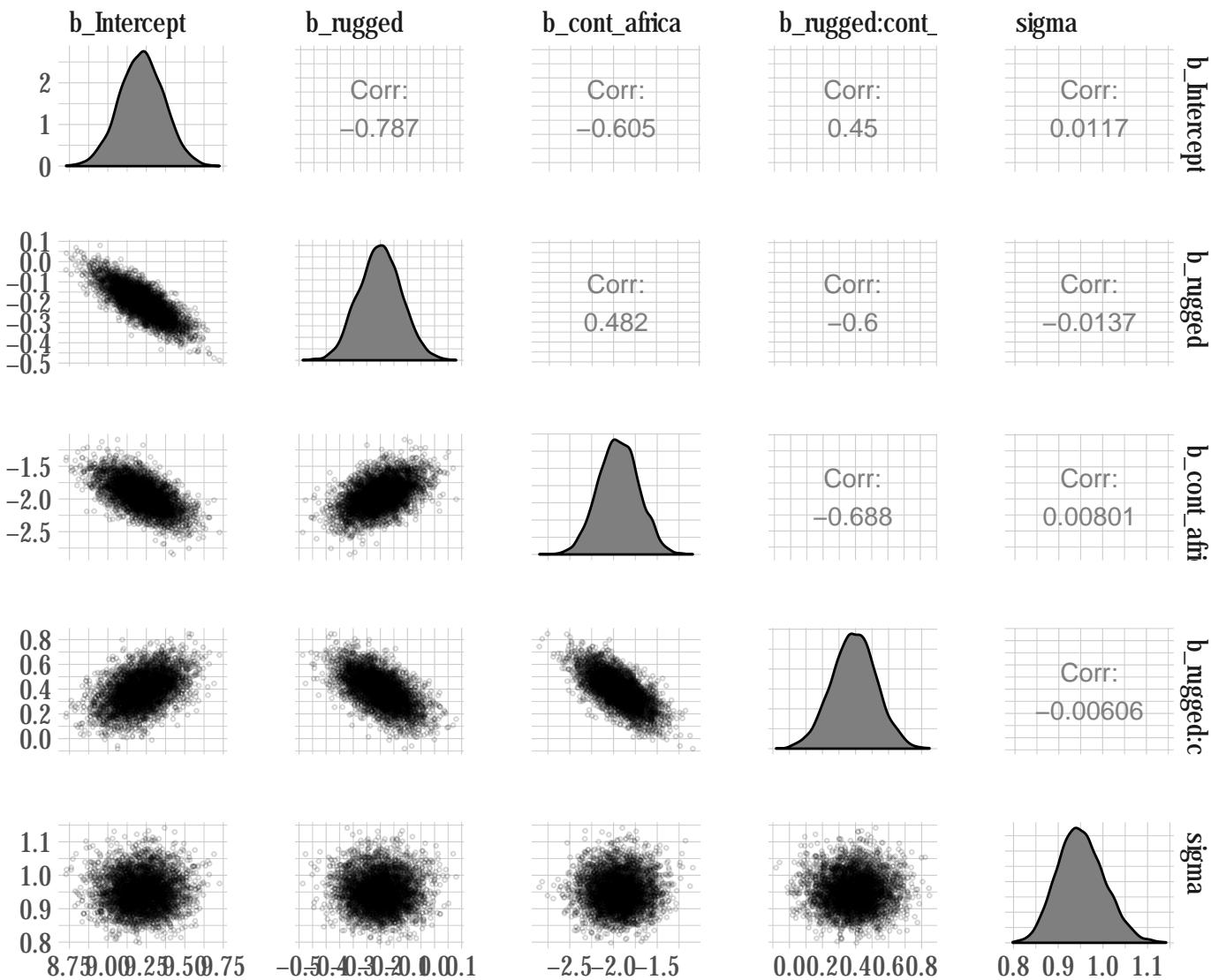
```
my_diag <- function(data, mapping, ...){
  ggplot(data = data, mapping = mapping) +
    geom_density(fill = "grey50")
}

my_lower <- function(data, mapping, ...){
  ggplot(data = data, mapping = mapping) +
    geom_point(shape = 1, size = 1/2, alpha = 1/6)
}

post %>%
  select(b_Intercept:sigma) %>%

  ggpairs(diag = list(continuous = my_diag),
         lower = list(continuous = my_lower)) +
  labs(subtitle = "My custom pairs plot") +
  theme_ipsum()
```

My custom pairs plot



For more ideas on customizing a GGally pairs plot, go [here](#).

8.3.5 Using the samples.

Older versions of brms allowed users to include information criteria as a part of the model summary by adding `loo = T` and/or `waic = T` in the `summary()` function (e.g., `summary(b8.1, loo = T, waic = T)`). However, this is no longer the case. E.g.,

```
summary(b8.1, loo = T, waic = T)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: log_gdp ~ 1 + rugged + cont_africa + rugged:cont_africa
##   Data: dd (Number of observations: 170)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##                               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  9.22      0.14     8.94     9.50       2792 1.00
## rugged                     -0.20      0.08    -0.35    -0.05       2782 1.00
```

```

## cont_africa      -1.94      0.23     -2.39     -1.50      2900 1.00
## rugged:cont_africa    0.39      0.13      0.12      0.65      2741 1.00
##
## Family Specific Parameters:
##   Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma     0.95      0.05      0.86      1.06      4183 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Although R didn't bark at us for adding `loo = T`, `waic = T`, they didn't do anything. Nowadays, if you want that information, you'll have to use the `waic()` and/or `loo()` functions.

```
waic(b8.1)
```

```

##
## Computed from 4000 by 170 log-likelihood matrix
##
##           Estimate   SE
## elpd_waic   -234.8  7.4
## p_waic       5.2   0.9
## waic        469.5 14.8
##
## Warning: 2 (1.2%) p_waic estimates greater than 0.4. We recommend trying
## loo instead.

```

```
(l_b8.1 <- loo(b8.1))
```

```

##
## Computed from 4000 by 170 log-likelihood matrix
##
##           Estimate   SE
## elpd_loo   -234.8  7.4
## p_loo       5.3   0.9
## looic      469.7 14.8
## -----
## Monte Carlo SE of elpd_loo is 0.0.
##
## All Pareto k estimates are good (k < 0.5).
## See help('pareto-k-diagnostic') for details.

```

And the recommended workflow since brms version 2.8.0 is to save the information criteria information with your `brm()` fit objects with the `add_criterion()` function.

```
b8.1 <- add_criterion(b8.1, c("waic", "loo"))
```

You retrieve that information like this:

```
b8.1$waic
```

```

##
## Computed from 4000 by 170 log-likelihood matrix
##
##           Estimate   SE
## elpd_waic   -234.8  7.4
## p_waic       5.2   0.9
## waic        469.5 14.8

```

```
## Warning: 2 (1.2%) p_waic estimates greater than 0.4. We recommend trying
## loo instead.
```

```
b8.1$loo
```

```
##
## Computed from 4000 by 170 log-likelihood matrix
##
##          Estimate    SE
## elpd_loo   -234.8  7.4
## p_loo       5.3   0.9
## looic     469.7 14.8
## -----
## Monte Carlo SE of elpd_loo is 0.0.
##
## All Pareto k estimates are good (k < 0.5).
## See help('pareto-k-diagnostic') for details.
```

In response to the brms version 2.8.0 update, which itself accommodated updates to the loo package and both of which occurred years after McElreath published the first edition of his text, we've been bantering on about the elpd and its relation to the WAIC and the LOO since Chapter 6. This is a fine place to go into some detail.

The `elpd` values returned by `loo()` and `waic()` are the expected log pointwise predictive density for new data. It follows the formula

$$\text{elpd} = \sum_{i=1}^n \int p_t(\tilde{y}_i) \log p(\tilde{y}_i | y) d\tilde{y}_i,$$

where $p_t(\tilde{y}_i)$ is the distribution representing the true data-generating process for \tilde{y}_i . The $p_t(\tilde{y}_i)$'s are unknown, and we will use cross-validation or WAIC to approximate. In a regression, these distributions are also implicitly conditioned on any predictors in the model. ([Vehtari, Gelman, & Gabry, 2016, p. 2](#)).

Later in the paper, we learn the `elpd_loo` (i.e., the Bayesian LOO estimate of out-of-sample predictive fit) is defined as

$$\text{elpd}_{\text{loo}} = \sum_{i=1}^n \log p(y_i | y_{-i}),$$

where

$$p(y_i | y_{-i}) = \int p(y_i | \theta) p(\theta | y_{-i}) d\theta$$

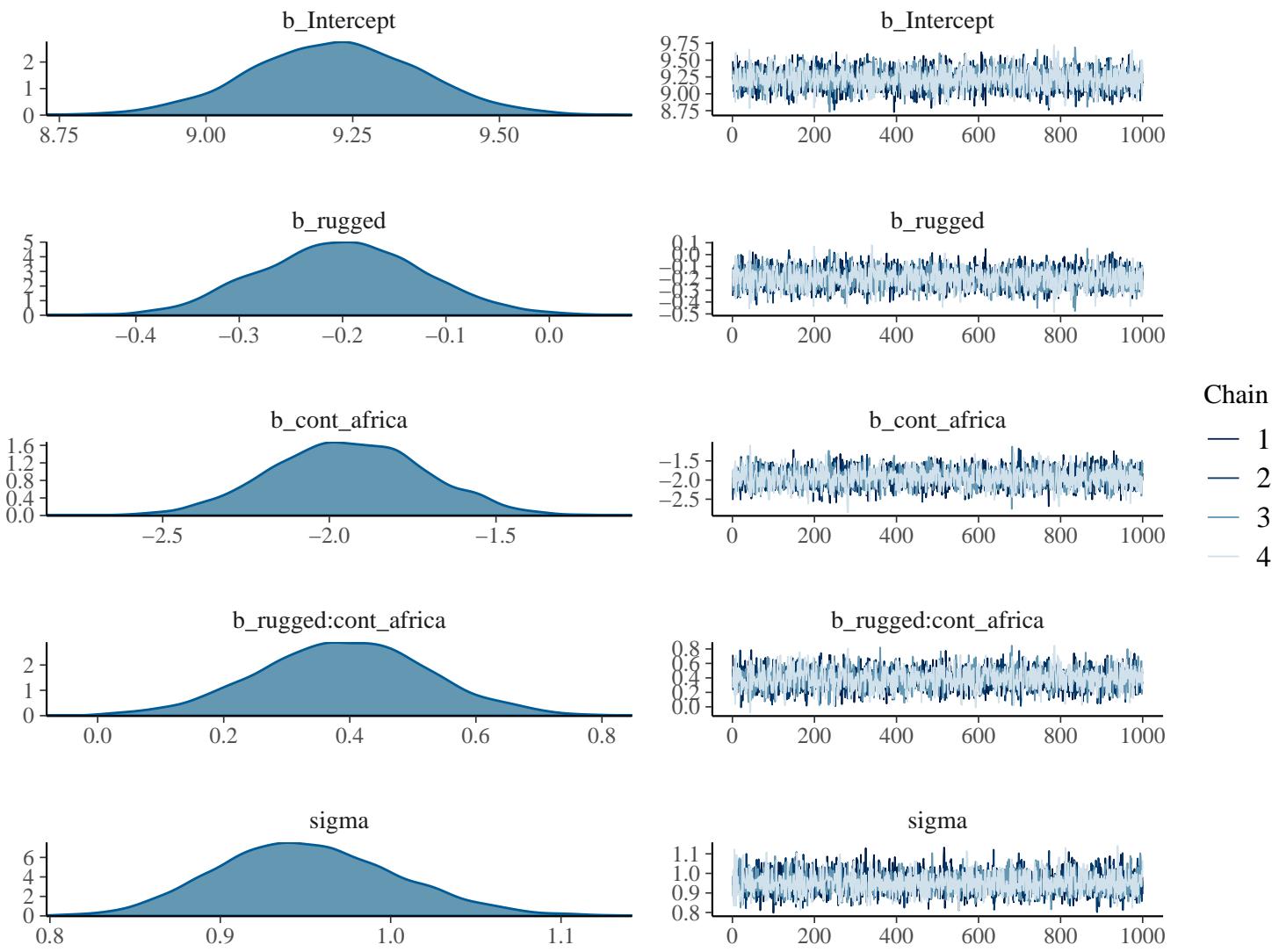
“is the leave-one-out predictive density given the data without the i th data point” (p. 3). And recall, you can convert the elpd to the conventional information criteria metric by multiplying it by -2.

To learn more about the elpd, read the rest of the paper and the [other works referenced by the loo package team](#). And if you prefer watching video lectures to reading technical papers, check out Vehtari's [Model assessment, selection and averaging](#).

8.3.6 Checking the chain.

Using `plot()` for a `brm()` fit returns both density and trace lots for the parameters.

```
plot(b8.1)
```



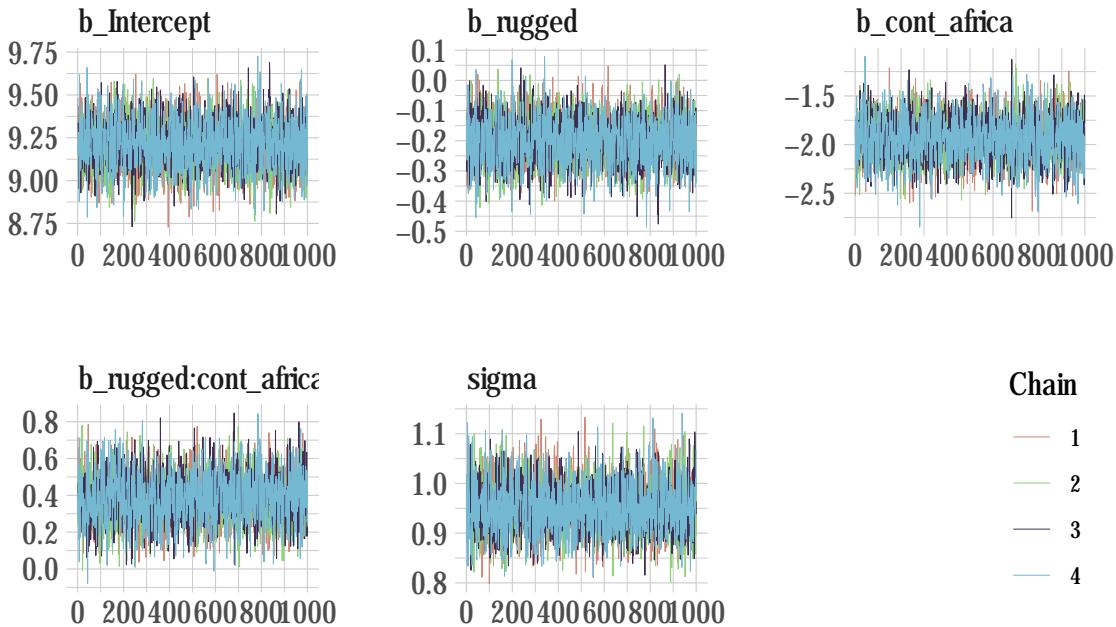
The `bayesplot` package allows a little more control. Here, we use `bayesplot's` `mcmc_trace()` to show only trace plots with our custom theme. Note that `mcmc_trace()` works with data frames, not `brmfit` objects. There's a further complication. Recall how we made `post` (i.e., `post <- posterior_samples(b8.1)`). Our `post` data frame carries no information on chains. To retain that information, we'll need to add an `add_chain = T` argument to our `posterior_samples()` function.

```
library(bayesplot)

post <- posterior_samples(b8.1, add_chain = T)

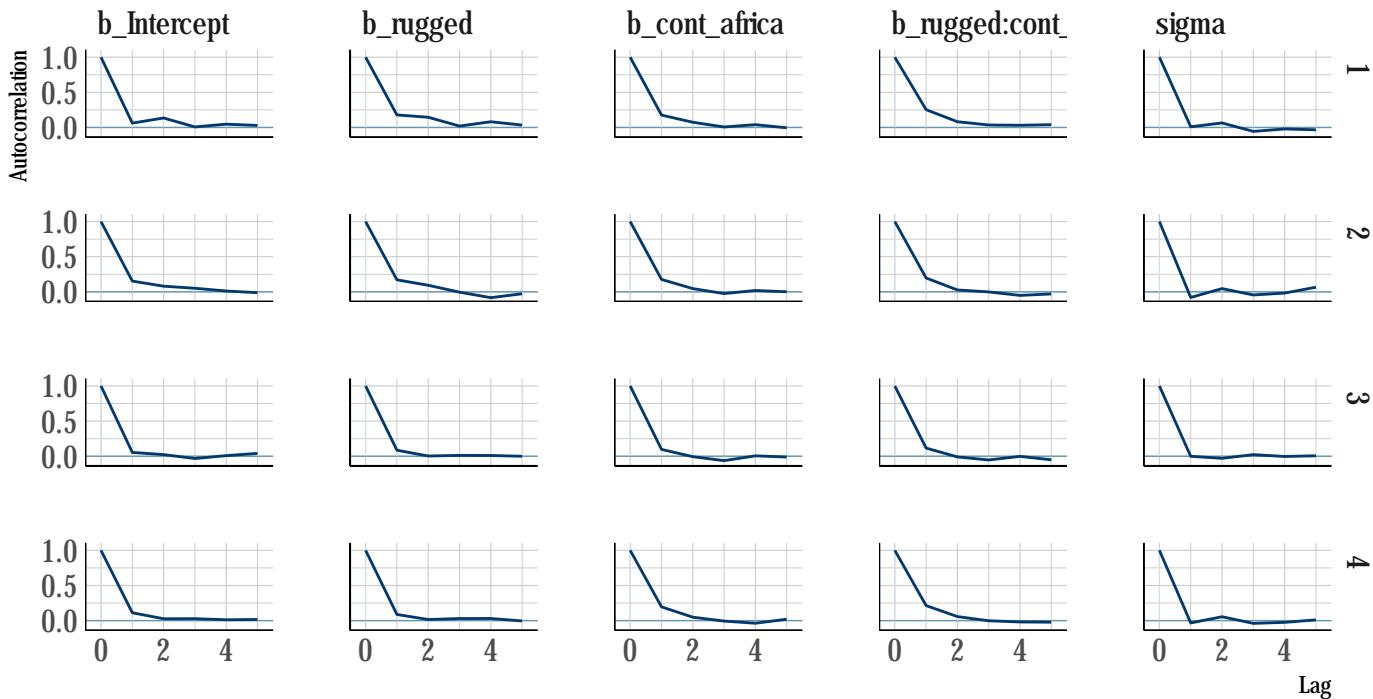
mcmc_trace(post[, c(1:5, 7)], # we need to include column 7 because it contains the chain info
           facet_args = list(ncol = 3),
           size = .15) +
  labs(title = "My custom trace plots") +
  scale_color_ipsum() +
  theme_ipsum() +
  theme(legend.position = c(.95, .2))
```

My custom trace plots



The bayesplot package offers a variety of diagnostic plots. Here we make autocorrelation plots for all model parameters, one for each HMC chain.

```
mcmc_acf(post,
  pars = c("b_Intercept", "b_rugged", "b_cont_africa", "b_rugged:cont_africa", "sigma"),
  lags = 5) +
  scale_color_ipsum() +
  theme_ipsum()
```



That's just what we like to see—nice L-shaped autocorrelation plots. Those are the kinds of shapes you'd expect when you have reasonably large effective samples. Anyway...

8.3.6.1 Overthinking: Raw Stan model code.

The `stancode()` function works in brms much like it does in rethinking.

```
brms::stancode(b8.1)
```

```
## // generated with brms 2.8.0
## functions {
## }
## data {
##   int<lower=1> N;    // number of observations
##   vector[N] Y;      // response variable
##   int<lower=1> K;    // number of population-level effects
##   matrix[N, K] X;   // population-level design matrix
##   int prior_only;   // should the likelihood be ignored?
## }
## transformed data {
##   int Kc = K - 1;
##   matrix[N, K - 1] Xc; // centered version of X
##   vector[K - 1] means_X; // column means of X before centering
##   for (i in 2:K) {
##     means_X[i - 1] = mean(X[, i]);
##     Xc[, i - 1] = X[, i] - means_X[i - 1];
##   }
## }
## parameters {
##   vector[Kc] b; // population-level effects
##   real temp_Intercept; // temporary intercept
##   real<lower=0> sigma; // residual SD
## }
## transformed parameters {
## }
## model {
##   vector[N] mu = temp_Intercept + Xc * b;
##   // priors including all constants
##   target += normal_lpdf(b | 0, 10);
##   target += normal_lpdf(temp_Intercept | 0, 100);
##   target += cauchy_lpdf(sigma | 0, 2)
##     - 1 * cauchy_lccdf(0 | 0, 2);
##   // likelihood including all constants
##   if (!prior_only) {
##     target += normal_lpdf(Y | mu, sigma);
##   }
## }
## generated quantities {
##   // actual population-level intercept
##   real b_Intercept = temp_Intercept - dot_product(means_X, b);
## }
```

You can also get that information with `b8.1$model` or `b8.1$fit@stanmodel`.

8.4 Care and feeding of your Markov chain.

Markov chain Monte Carlo is a highly technical and usually automated procedure. Most people who use it don't really understand what it is doing. That's okay, up to a point. Science requires division of labor, and if every one of us had to write our own Markov chains from scratch, a lot less research would get done in the aggregate. (p. 255)

But if you do want to learn more about HMC, McElreath has some nice introductory lectures on the topic (see [here](#) and [here](#)). To dive even deeper, [Michael Betancourt](#) from the Stan team has given many lectures on the topic (e.g., [here](#) and [here](#)).

8.4.1 How many samples do you need?

The brms defaults for `iter` and `warmup` match those of McElreath's rethinking.

If all you want are posterior means, it doesn't take many samples at all to get very good estimates. Even a couple hundred samples will do. But if you care about the exact shape in the extreme tails of the posterior, the 99th percentile or so, then you'll need many many more. So there is no universally useful number of samples to aim for. In most typical regression applications, you can get a very good estimate of the posterior mean with as few as 200 effective samples. And if the posterior is approximately Gaussian, then all you need in addition is a good estimate of the variance, which can be had with one order of magnitude more, in most cases. For highly skewed posteriors, you'll have to think more about which region of the distribution interests you. (p. 255)

8.4.2 How many chains do you need?

"Using 3 or 4 chains is conventional, and quite often more than enough to reassure us that the sampling is working properly" (p. 257).

8.4.2.1 Convergence diagnostics.

The default diagnostic output from Stan includes two metrics, `n_eff` and `Rhat`. The first is a measure of the effective number of samples. The second is the Gelman-Rubin convergence diagnostic, \hat{R} . When `n_eff` is much lower than the actual number of iterations (minus warmup) of your chains, it means the chains are inefficient, but possibly still okay. When `Rhat` is above 1.00, it usually indicates that the chain has not yet converged, and probably you shouldn't trust the samples. If you draw more iterations, it could be fine, or it could never converge. See the [Stan user manual](#) for more details. It's important however not to rely too much on these diagnostics. Like all heuristics, there are cases in which they provide poor advice. (p. 257)

For more on `n_eff` and `Rhat`, you might also check out Gabry and Modrák's vignette, [Visual MCMC diagnostics using the bayesplot package](#).

The \hat{R} has been our friend for many years. But times are changing. As it turns out, the Stan team has found some deficiencies with the \hat{R} , for which they've made recommendations that will be implemented in the Stan ecosystem sometime soon. In the meantime, you can read all about it in their [preprint](#) and in one of Dan Simpson's [blogs](#). If you learn best by sassy twitter banter, [click through this interchange](#) among some of our Stan team all-stars.

8.4.3 Taming a wild chain.

As with rethinking, brms can take data in the form of a list. Recall however, that in order to specify starting values, you need to specify a list of lists with an `inits` argument rather than with `start`.

```
b8.2 <-  
  brm(data = list(y = c(-1, 1)),  
       family = gaussian,  
       y ~ 1,  
       prior = c(prior(uniform(-1e10, 1e10), class = Intercept),  
                 prior(uniform(0, 1e10), class = sigma)),  
       inits = list(list(Intercept = 0, sigma = 1),  
                  list(Intercept = 0, sigma = 1)),  
       iter = 4000, warmup = 1000, chains = 2,  
       seed = 8)
```

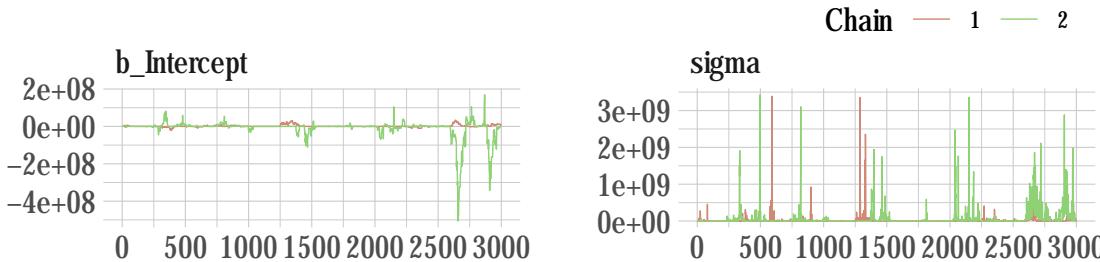
Those were some silly flat priors. Check the damage.

```
post <- posterior_samples(b8.2, add_chain = T)  
  
mcmc_trace(post[, c(1:2, 4)],  
            size = .25) +
```

```
labs(title = "My version of Figure 8.5.a.",
     subtitle = "These trace plots do not look like the fuzzy caterpillars we usually hope for.") +
scale_color_ipsum() +
theme_ipsum() +
theme(legend.position = c(.85, 1.5),
      legend.direction = "horizontal")
```

My version of Figure 8.5.a.

These trace plots do not look like the fuzzy caterpillars we usually hope for.



Let's peek at the summary.

```
print(b8.2)
```

```
## Warning: There were 609 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help.
## See http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: y ~ 1
## Data: list(y = c(-1, 1)) (Number of observations: 2)
## Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate   Est.Error 1-95% CI  u-95% CI Eff.Sample Rhat
## Intercept -3865553.82 35800459.25 -74287051.02 24607826.73           105 1.04
##
## Family Specific Parameters:
##             Estimate   Est.Error 1-95% CI  u-95% CI Eff.Sample Rhat
## sigma    40602239.10 188456908.77  3167.70 376813947.39           266 1.01
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Holy smokes, those parameters are a mess! Plus we got a nasty warning message, too. Watch our reasonable priors save the day.

```
b8.3 <-
brm(data = list(y = c(-1, 1)),
     family = gaussian,
     y ~ 1,
     prior = c(prior(normal(0, 10), class = Intercept),
               prior(cauchy(0, 1), class = sigma)),
     inits = list(list(Intercept = 0, sigma = 1),
                 list(Intercept = 0, sigma = 1)),
     iter = 4000, warmup = 1000, chains = 2,
     seed = 8)
```

```
print(b8.3)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: y ~ 1
##   Data: list(y = c(-1, 1)) (Number of observations: 2)
## Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     -0.01      1.61    -3.38     3.46        1408 1.01
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       2.01      2.21     0.60     6.66        1668 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

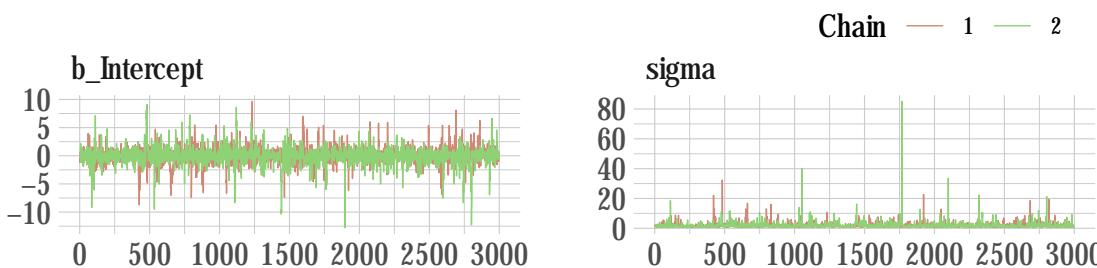
As in the text, no more warning signs and no more silly estimates. The trace plots look great, too.

```
post <- posterior_samples(b8.3, add_chain = T)

mcmc_trace(post[, c(1:2, 4)],
            size = .25) +
  labs(title      = "My version of Figure 8.5.b",
       subtitle    = "Oh man. This looks so much better.") +
  scale_color_ipsum() +
  theme_ipsum() +
  theme(legend.position = c(.85, 1.5),
        legend.direction = "horizontal")
```

My version of Figure 8.5.b

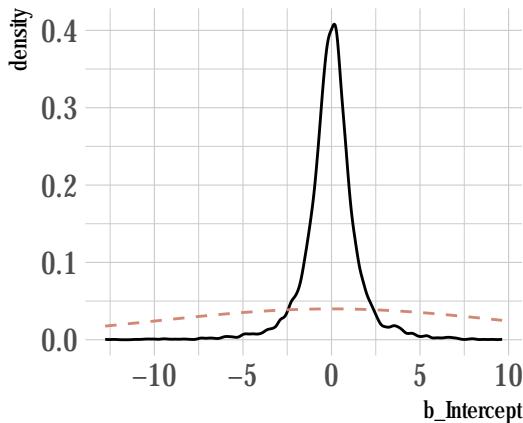
Oh man. This looks so much better.



Now behold our version of Figure 8.6.a.

```
post %>%
  select(b_Intercept) %>%

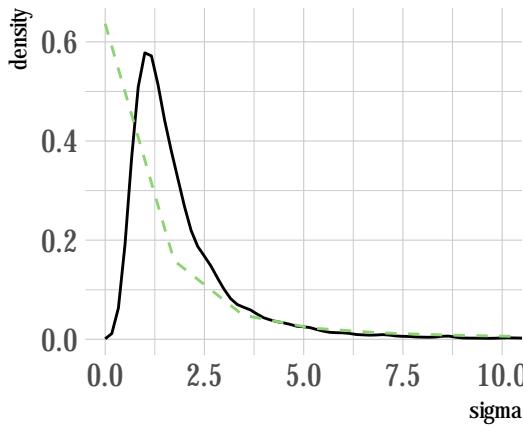
  ggplot(aes(x = b_Intercept)) +
  stat_density(geom = "line") +
  geom_line(data = data.frame(x = seq(from = min(post$b_Intercept),
                                         to = max(post$b_Intercept),
                                         length.out = 50)),
            aes(x = x, y = dnorm(x = x, mean = 0, sd = 10)),
            color = ipsum_pal()(1), linetype = 2) +
  theme_ipsum()
```



Here's our version of Figure 8.6.b.

```
post %>%
  select(sigma) %>%

  ggplot(aes(x = sigma)) +
  stat_density(geom = "line") +
  geom_line(data = data.frame(x = seq(from = 0,
                                         to = max(post$sigma),
                                         length.out = 50)),
            aes(x = x, y = dcauchy(x = x, location = 0, scale = 1)*2),
            color = ipsum_pal()(2)[2], linetype = 2) +
  coord_cartesian(xlim = c(0, 10)) +
  theme_ipsum()
```



8.4.3.1 Overthinking: Cauchy distribution.

Behold the beautiful Cauchy probability density:

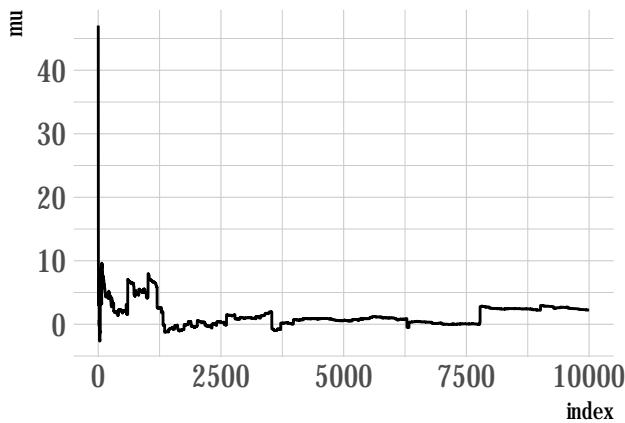
$$p(x|x_0, \gamma) = \left(\pi \gamma \left[1 + \left(\frac{x - x_0}{\gamma} \right)^2 \right] \right)^{-1}$$

The Cauchy has no mean and variance, but x_0 is the location and γ is the scale. Here's our version of the simulation. Note our use of the `cummean()` function.

```
n <- 1e4

set.seed(8)
tibble(y      = rcauchy(n, location = 0, scale = 5),
       mu     = cummean(y),
```

```
index = 1:n) %>%
ggplot(aes(x = index, y = mu)) +
geom_line() +
theme_ipsum()
```

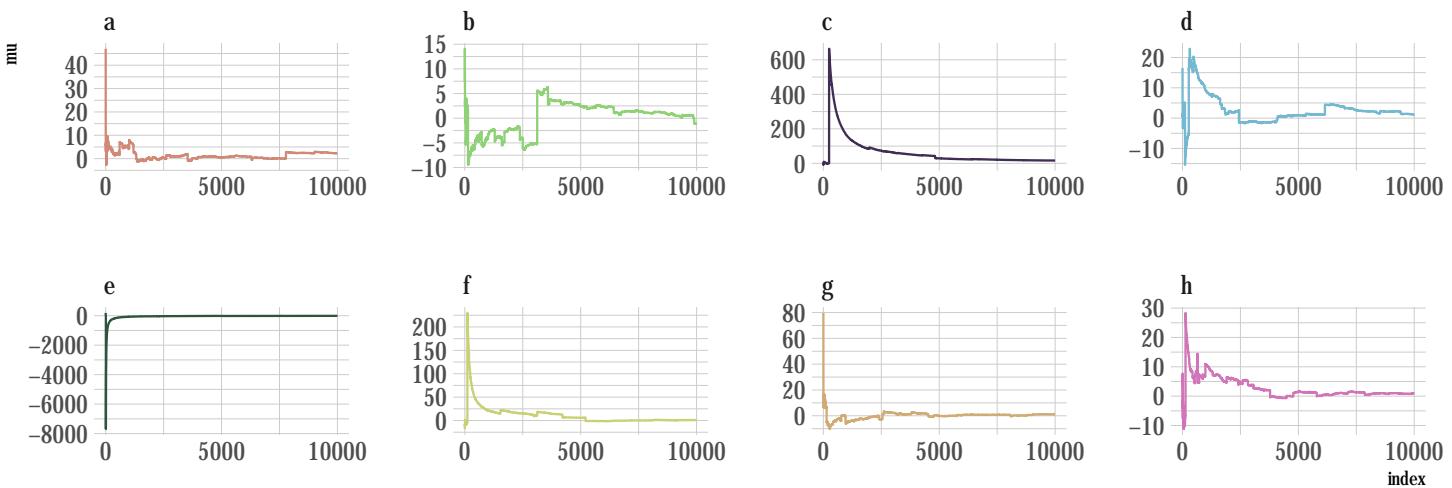


The whole thing is quite remarkable. Just for kicks, here we do it again, this time with eight simulations.

```
n <- 1e4

set.seed(8)
tibble(a = rcauchy(n, location = 0, scale = 5),
       b = rcauchy(n, location = 0, scale = 5),
       c = rcauchy(n, location = 0, scale = 5),
       d = rcauchy(n, location = 0, scale = 5),
       e = rcauchy(n, location = 0, scale = 5),
       f = rcauchy(n, location = 0, scale = 5),
       g = rcauchy(n, location = 0, scale = 5),
       h = rcauchy(n, location = 0, scale = 5)) %>%
gather() %>%
group_by(key) %>%
mutate(mu = cummean(value)) %>%
ungroup() %>%
mutate(index = rep(1:n, times = 8)) %>%

ggplot(aes(x = index, y = mu)) +
geom_line(aes(color = key)) +
scale_color_manual(values = ipsum_pal()(8)) +
scale_x_continuous(breaks = c(0, 5000, 10000)) +
theme_ipsum() +
theme(legend.position = "none") +
facet_wrap(~key, ncol = 4, scales = "free")
```



8.4.4 Non-identifiable parameters.

It appears that the [only way](#) to get a brms version of McElreath's `m8.4` and `m8.5` is to augment the data. In addition to the Gaussian `y` vector, we'll add two constants to the data, `intercept_1 = 1` and `intercept_2 = 1`.

```
set.seed(8)
y <- rnorm(100, mean = 0, sd = 1)
```

```
b8.4 <-
  brm(data = list(y           = y,
                   intercept_1 = 1,
                   intercept_2 = 1),
       family = gaussian,
       y ~ 0 + intercept_1 + intercept_2,
       prior = c(prior(uniform(-1e10, 1e10), class = b),
                  prior(cauchy(0, 1), class = sigma)),
       inits = list(list(intercept_1 = 0, intercept_2 = 0, sigma = 1),
                    list(intercept_1 = 0, intercept_2 = 0, sigma = 1)),
       iter = 4000, warmup = 1000, chains = 2,
       seed = 8)
```

Our model results don't perfectly mirror McElreath's, but they're identical in spirit.

```
print(b8.4)
```

```
## Warning: The model has not converged (some Rhats are > 1.1). Do not analyse the results!
## We recommend running more iterations and/or setting stronger priors.

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: y ~ 0 + intercept_1 + intercept_2
##   Data: list(y = y, intercept_1 = 1, intercept_2 = 1) (Number of observations: 100)
##   Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##             total post-warmup samples = 6000
##
## Population-Level Effects:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## intercept_1  1116.48    944.73  -762.62  2439.62        2  2.50
## intercept_2 -1116.57    944.74 -2439.65   762.38        2  2.50
##
## Family Specific Parameters:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
```

```
## sigma      1.06      0.08      0.91      1.22      9 1.16
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Note the frightening warning message. Those results are a mess! Let's try again.

```
b8.5 <-
  brm(data = list(y           = y,
                   intercept_1 = 1,
                   intercept_2 = 1),
       family = gaussian,
       y ~ 0 + intercept_1 + intercept_2,
       prior = c(prior(normal(0, 10), class = b),
                  prior(cauchy(0, 1), class = sigma)),
       inits = list(list(intercept_1 = 0, intercept_2 = 0, sigma = 1),
                    list(intercept_1 = 0, intercept_2 = 0, sigma = 1)),
       iter = 4000, warmup = 1000, chains = 2,
       seed = 8)
```

```
print(b8.5)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: y ~ 0 + intercept_1 + intercept_2
##   Data: list(y = y, intercept_1 = 1, intercept_2 = 1) (Number of observations: 100)
## Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## intercept_1    -0.11     7.04   -14.32    13.56      1357 1.00
## intercept_2     0.02     7.04   -13.59    14.26      1359 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.09     0.08     0.95     1.25      2086 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Much better. Now we'll do the preparatory work for Figure 8.7. Instead of showing the plots, here, we'll save them as objects, `left_column` and `right_column`, in order to combine them below.

```
post <- posterior_samples(b8.4, add_chain = T)

left_column <-
  mcmc_trace(post[, c(1:3, 5)],
              size = .25,
              facet_args = c(ncol = 1)) +
  scale_color_ipsum() +
  theme_ipsum() +
  theme(legend.position = c(.85, 1.5),
        legend.direction = "horizontal")

post <- posterior_samples(b8.5, add_chain = T)
```

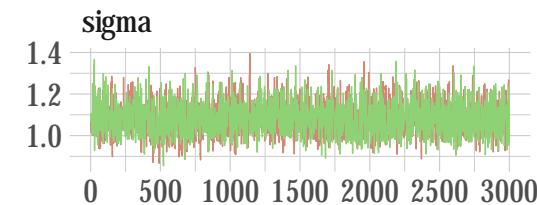
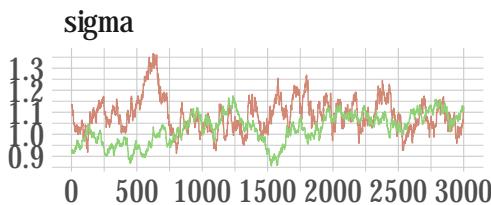
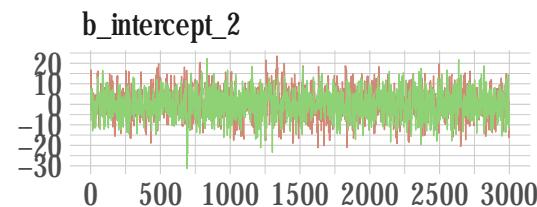
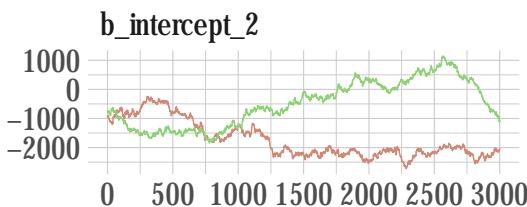
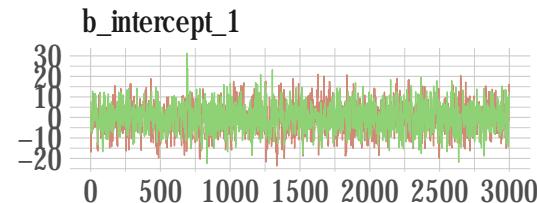
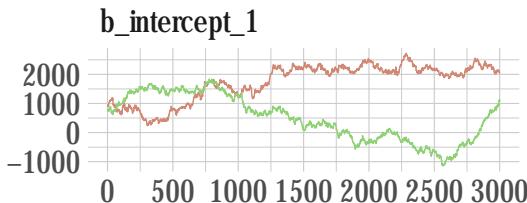
```

right_column <-
  mcmc_trace(post[, c(1:3, 5)],
    size = .25,
    facet_args = c(ncol = 1)) +
  scale_color_ipsum() +
  theme_ipsum() +
  theme(legend.position = c(.85, 1.5),
    legend.direction = "horizontal")

library(gridExtra)

grid.arrange(left_column, right_column, ncol = 2)

```



The central message in the text, default to weakly-regularizing priors, holds for brms just as it does in rethinking. For more on the topic, see the [recommendations from the Stan team](#). If you want to dive deeper, check out [Dan Simpson's post on Gelman's blog](#) and their [corresponding paper](#) with Michael Betancourt.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```

sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default

```

```
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel   stats      graphics   grDevices  utils      datasets   methods
## [8] base
##
## other attached packages:
## [1] gridExtra_2.3      bayesplot_1.6.0    GGally_1.4.0
## [4] tidybayes_1.0.4    brms_2.8.8        Rcpp_1.0.1
## [7] rstan_2.18.2       StanHeaders_2.18.0-1 forcats_0.3.0
## [10] stringr_1.4.0     dplyr_0.8.0.1    purrr_0.2.5
## [13] readr_1.1.1       tidyverse_1.2.1   tibble_2.1.1
## [16] ggplot2_3.1.1     tidyverse_1.2.1   extrafont_0.17
## [19] hrbrthemes_0.6.0
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2      ggridges_0.5.0
## [3] rsconnect_0.8.8       rprojroot_1.3-2
## [5] ggstance_0.3          markdown_0.8
## [7] base64enc_0.1-3       rethinking_1.80
## [9] rstudioapi_0.7        svUnit_0.7-12
## [11] DT_0.4                fansi_0.4.0
## [13] mvtnorm_1.0-10       lubridate_1.7.4
## [15] xml2_1.2.0            bridgesampling_0.6-0
## [17] knitr_1.20             shinythemes_1.1.1
## [19] jsonlite_1.5          broom_0.5.1
## [21] Rttf2pt1_1.3.7        shiny_1.1.0
## [23] compiler_3.5.1        httr_1.3.1
## [25] backports_1.1.4       assertthat_0.2.0
## [27] Matrix_1.2-14         lazyeval_0.2.2
## [29] cli_1.0.1              later_0.7.3
## [31] htmltools_0.3.6        prettyunits_1.0.2
## [33] tools_3.5.1            igraph_1.2.1
## [35] coda_0.19-2           gtable_0.3.0
## [37] glue_1.3.1.9000        reshape2_1.4.3
## [39] cellranger_1.1.0       nlme_3.1-137
## [41] extrafontdb_1.0         crosstalk_1.0.0
## [43] xfun_0.3                ps_1.2.1
## [45] rvest_0.3.2             mime_0.5
## [47] miniUI_0.1.1.1         gtools_3.8.1
## [49] MASS_7.3-50              zoo_1.8-2
## [51] scales_1.0.0            colourpicker_1.0
## [53] hms_0.4.2                promises_1.0.1
## [55] Brobdingnag_1.2-6       inline_0.3.15
## [57] RColorBrewer_1.1-2       shinystan_2.5.0
## [59] yaml_2.1.19              gdtools_0.1.7
## [61] loo_2.1.0                 reshape_0.8.7
## [63] stringi_1.4.3            dygraphs_1.1.1.5
## [65] pkgbuild_1.0.2            rlang_0.3.4
## [67] pkgconfig_2.0.2           matrixStats_0.54.0
## [69] HDInterval_0.2.0          evaluate_0.10.1
## [71] lattice_0.20-35          rstantools_1.5.1
## [73] htmlwidgets_1.2            labeling_0.3
## [75] processx_3.2.1           tidyselect_0.2.5
## [77] plyr_1.8.4                magrittr_1.5
## [79] bookdown_0.9               R6_2.3.0
```

```
## [81] generics_0.0.2          pillar_1.3.1
## [83] haven_1.1.2            withr_2.1.2
## [85] xts_0.10-2             abind_1.4-5
## [87] modelr_0.1.2           crayon_1.3.4
## [89] arrayhelpers_1.0-20160527 utf8_1.1.4
## [91] rmarkdown_1.10           grid_3.5.1
## [93] readxl_1.1.0            callr_3.1.0
## [95] threejs_0.3.1           digest_0.6.18
## [97] xtable_1.8-2            httpuv_1.4.4.2
## [99] stats4_3.5.1            munsell_0.5.0
## [101] shinyjs_1.0
```


Chapter 9

Big Entropy and the Generalized Linear Model

... Statistical models force many choices upon us. Some of these choices are distributions that represent uncertainty. We must choose, for each parameter, a prior distribution. And we must choose a likelihood function, which serves as a distribution of data. There are conventional choices, such as wide Gaussian priors and the Gaussian likelihood of linear regression. These conventional choices work unreasonably well in many circumstances. But very often the conventional choices are not the best choices. Inference can be more powerful when we use all of the information, and doing so usually requires going beyond convention.

To go beyond convention, it helps to have some principles to guide choice. When an engineer wants to make an unconventional bridge, engineering principles help guide choice. When a researcher wants to build an unconventional model, entropy provides one useful principle to guide choice of probability distributions: Bet on the distribution with the biggest entropy. (p. 267)

9.1 Maximum entropy

In Chapter 6, you met the basics of information theory. In brief, we seek a measure of uncertainty that satisfies three criteria: (1) the measure should be continuous; (2) it should increase as the number of possible events increases; and (3) it should be additive. The resulting unique measure of the uncertainty of a probability distribution p with probabilities p_i for each possible event i turns out to be just the average log-probability:

$$H(p) = - \sum_i p_i \log p_i$$

This function is known as *information entropy*. (p. 268, *emphasis* in the original)

Let's execute the code for the pebbles-in-buckets example.

```
library(tidyverse)

d <-
  tibble(a = c(0, 0, 10, 0, 0),
        b = c(0, 1, 8, 1, 0),
        c = c(0, 2, 6, 2, 0),
        d = c(1, 2, 4, 2, 1),
        e = 2)

# this is our analogue to McElreath's `lapply()` code
d %>%
  mutate_all(~ . / sum(.)) %>%
# the next few lines constitute our analogue to his `sapply()` code
  gather() %>%
```

```
group_by(key) %>%
  summarise(h = -sum(ifelse(value == 0, 0, value * log(value))))
```

```
## # A tibble: 5 x 2
##   key      h
##   <chr> <dbl>
## 1 a      0
## 2 b     0.639
## 3 c     0.950
## 4 d     1.47
## 5 e     1.61
```

For more on the formula syntax we used within `mutate_all()`, you might check out [this](#) or [this](#).

Anyway, we're almost ready to plot. Which brings us to color. For the plots in this chapter, we'll be taking our color palettes from the [ghibli package](#), which provides palettes based on scenes from anime films by the Studio Ghibli.

```
# install.packages("ghibli", dependencies = T)
library(ghibli)
```

The main function is `ghibli_palette()` which you can use to both preview the palettes before using them and also index in order to use specific colors. For example, we'll play with "MarnieMedium1", first.

```
ghibli_palette("MarnieMedium1")
```

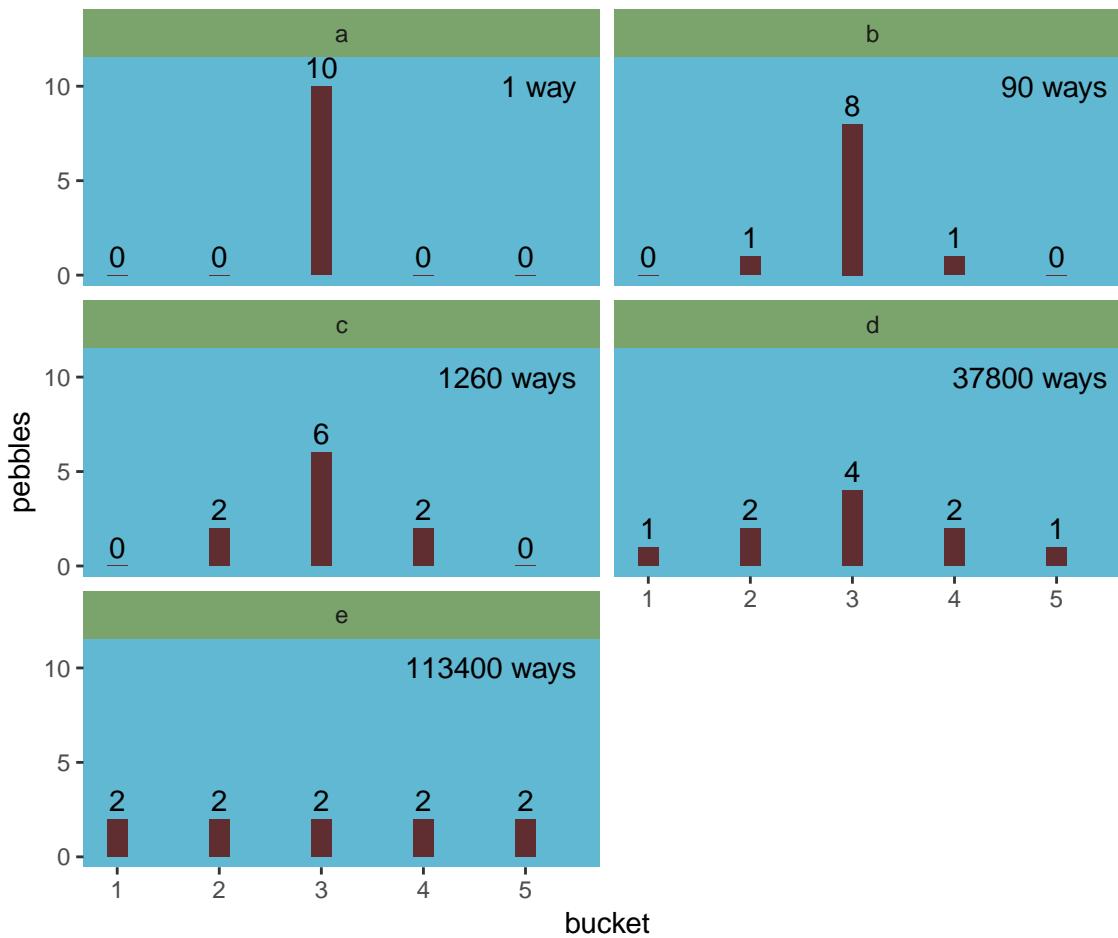


```
ghibli_palette("MarnieMedium1")[1:7]
```

```
## [1] "#7BA46C" "#602D31" "#008D91" "#0A789F" "#C6A28A" "#61B8D3" "#EACF9E"
```

Now we're ready to plot five of the six panels of Figure 9.1.

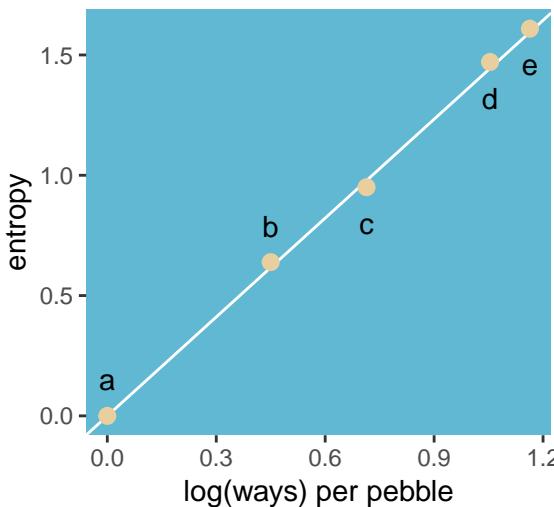
```
d %>%
  mutate(bucket = 1:5) %>%
  gather(letter, pebbles, - bucket) %>%
  ggplot(aes(x = bucket, y = pebbles)) +
  geom_col(width = 1/5, fill = ghibli_palette("MarnieMedium1")[2]) +
  geom_text(aes(y = pebbles + 1, label = pebbles)) +
  geom_text(data = tibble(
    letter = letters[1:5],
    bucket = 5.5,
    pebbles = 10,
    label = str_c(c(1, 90, 1260, 37800, 113400),
                 rep(c(" way", " ways"), times = c(1, 4))),
    aes(label = label), hjust = 1) +
  scale_y_continuous(breaks = c(0, 5, 10)) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("MarnieMedium1")[6]),
        strip.background = element_rect(fill = ghibli_palette("MarnieMedium1")[1])) +
  facet_wrap(~letter, ncol = 2)
```



We might plot the final panel like so.

```
d %>%
  # the next four lines are the same from above
  mutate_all(~ . / sum(.)) %>%
  gather() %>%
  group_by(key) %>%
  summarise(h = -sum(ifelse(value == 0, 0, value * log(value)))) %>%
  # here's the R code 9.4 stuff
  mutate(n_ways = c(1, 90, 1260, 37800, 113400)) %>%
  group_by(key) %>%
  mutate(log_ways = log(n_ways) / 10,
        text_y = ifelse(key < "c", h + .15, h - .15)) %>%

  # plot
  ggplot(aes(x = log_ways, y = h)) +
  geom_abline(intercept = 0, slope = 1.37,
              color = "white") +
  geom_point(size = 2.5, color = ghibli_palette("MarnieMedium1")[7]) +
  geom_text(aes(y = text_y, label = key)) +
  labs(x = "log(ways) per pebble",
       y = "entropy") +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("MarnieMedium1")[6]))
```



“The distribution that can happen the greatest number of ways is the most plausible distribution. Call this distribution the maximum entropy distribution” (p. 271). Among the pebbles, the maximum entropy distribution was e (i.e., the uniform).

9.1.1 Gaussian.

Behold the probability distribution for the generalized normal distribution:

$$\Pr(y|\mu, \alpha, \beta) = \frac{\beta}{2\alpha\Gamma\left(\frac{1}{\beta}\right)} e^{-\left(\frac{|y-\mu|}{\alpha}\right)^{\beta}}$$

In this formulation, α = the scale, β = the shape, μ = the location, and Γ = the [gamma function](#). If you read closely in the text, you’ll discover that the densities in the right panel of Figure 9.2 were all created with the constraint $\sigma^2 = 1$. But $\sigma^2 \neq \alpha$ and there’s no σ in the equations in the text. However, it appears the variance for the generalized normal distribution follows the form:

$$\sigma^2 = \frac{\alpha^2\Gamma(3/\beta)}{\Gamma(1/\beta)}$$

So if you do the algebra, you’ll see that you can compute α for a given σ^2 and β like so:

$$\alpha = \sqrt{\frac{\sigma^2\Gamma(1/\beta)}{\Gamma(3/\beta)}}$$

I got the formula from [Wikipedia.com](#). Don’t judge. We can wrap that formula in a custom function, `alpha_per_beta()`, use it to solve for the desired β values, and plot. But one more thing: McElreath didn’t tell us exactly which β values the left panel of Figure 9.2 was based on. So the plot below is my best guess.

```
alpha_per_beta <- function(variance, beta){
  sqrt((variance * gamma(1 / beta)) / gamma(3 / beta))
}

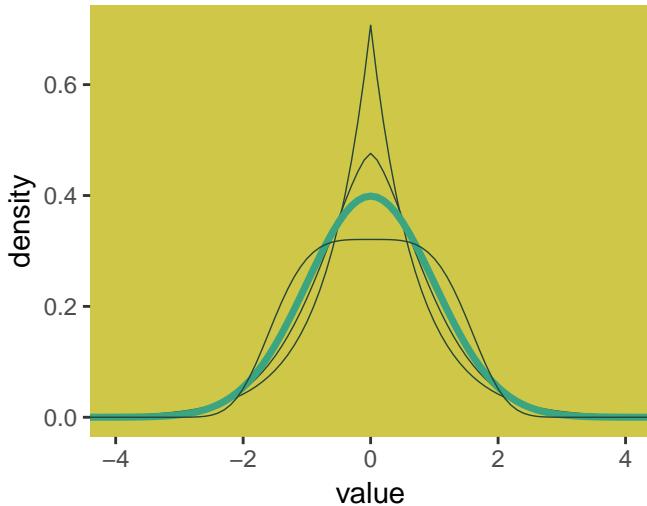
tibble(mu      = 0,
       variance = 1,
       # I arrived at these values by trial and error
       beta     = c(1, 1.5, 2, 4)) %>%
mutate(alpha = map2(variance, beta, alpha_per_beta)) %>%
unnest() %>%
expand(nesting(mu, beta, alpha),
       value = seq(from = -5, to = 5, by = .1)) %>%
```

```
# behold the formula for the generalized normal distribution in code
mutate(density = (beta / (2 * alpha * gamma(1 / beta))) *
       exp(1) ^ (-1 * (abs(value - mu) / alpha) ^ beta)) %>%

```

```
# plot
ggplot(aes(x = value, y = density,
            group = beta)) +
  geom_line(aes(color = beta == 2,
                size = beta == 2)) +
  scale_color_manual(values = c(ghibli_palette("MarnieMedium2")[2],
                               ghibli_palette("MarnieMedium2")[4])) +
  scale_size_manual(values = c(1/4, 1.25)) +
  ggtitle(NULL, subtitle = "Guess which color denotes the Gaussian.") +
  coord_cartesian(xlim = -4:4) +
  theme(panel.grid = element_blank(),
        legend.position = "none",
        panel.background = element_rect(fill = ghibli_palette("MarnieMedium2")[7]))
```

Guess which color denotes the Gaussian.

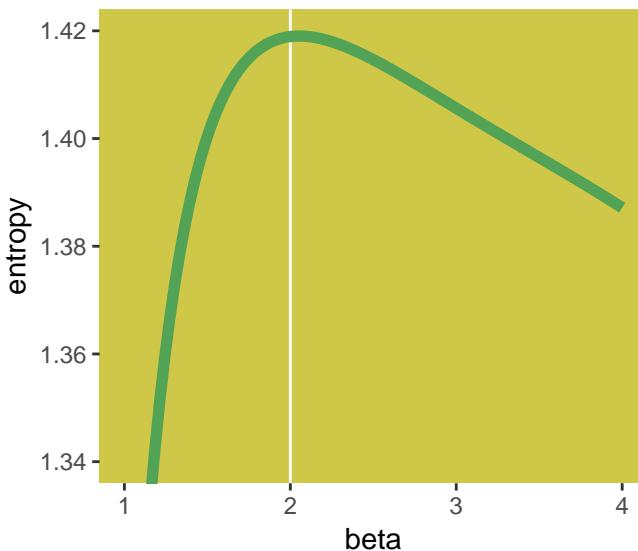


Here's Figure 9.2's right panel.

```
tibble(mu      = 0,
       variance = 1,
       # this time we need a more densely-packed sequence of `beta` values
       beta     = seq(from = 1, to = 4, length.out = 100)) %>%
  mutate(alpha = map2(variance, beta, alpha_per_beta)) %>%
  unnest() %>%
  expand(nesting(mu, beta, alpha),
         value = -8:8) %>%
  mutate(density = (beta / (2 * alpha * gamma(1 / beta))) *
         exp(1) ^ (-1 * (abs(value - mu) / alpha) ^ beta)) %>%
  group_by(beta) %>%
  # this is just an abbreviated version of the formula we used in our first code block
  summarise(entropy = -sum(density * log(density))) %>%

```

```
ggplot(aes(x = beta, y = entropy)) +
  geom_vline(xintercept = 2, color = "white") +
  geom_line(size = 2, color = ghibli_palette("MarnieMedium2")[6]) +
  coord_cartesian(ylim = c(1.34, 1.42)) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("MarnieMedium2")[7]))
```



If you look closely, you'll see our version doesn't quite match up with McElreath's. Over x-axis values of 2 to 4, they match up pretty well. But as you go from 2 to 1, you'll see our line drops off more steeply than his did. [And no, `coord_cartesian()` isn't the problem.] If you can figure out why our numbers diverged, [please share the answer](#).

But getting back on track:

The take-home lesson from all of this is that, if all we are willing to assume about a collection of measurements is that they have a finite variance, then the Gaussian distribution represents the most conservative probability distribution to assign to those measurements. But very often we are comfortable assuming something more. And in those cases, provided our assumptions are good ones, the principle of maximum entropy leads to distributions other than the Gaussian. (p. 274)

9.1.2 Binomial.

The binomial likelihood entails

counting the numbers of ways that a given observation could arise, according to assumptions... If only two things can happen (blue or white marble, for example), and there's a constant chance p of each across n trials, then the probability of observing y events of type 1 and $n - y$ events of type 2 is:

$$\Pr(y|n, p) = \frac{n!}{y!(n-y)!} p^y (1-p)^{n-y}$$

It may help to note that the fraction with the factorials is just saying how many different ordered sequences of n outcomes have a count of y . (p. 275)

For me, that last sentence made more sense when I walked it out in an example. To do so, lets wrap that fraction of factorials into a function.

```
count_ways <- function(n, y){
  # n = the total number of trials (i.e., the number of rows in your vector)
  # y = the total number of 1s (i.e., successes) in your vector
  (factorial(n) / (factorial(y) * factorial(n - y)))
}
```

Now consider three sequences:

- 0, 0, 0, 0 (i.e., $n = 4$ and $y = 0$)
- 1, 0, 0, 0 (i.e., $n = 4$ and $y = 1$)
- 1, 1, 0, 0 (i.e., $n = 4$ and $y = 2$)

We can organize that information in a little tibble and then demo our `count_ways()` function.

```
tibble(sequence = 1:3,
       n      = 4,
       y      = c(0, 1, 2)) %>%
  mutate(n_ways = map2(n, y, count_ways)) %>%
  unnest()
```

```
## # A tibble: 3 x 4
##   sequence     n     y n_ways
##       <int> <dbl> <dbl> <dbl>
## 1         1     4     0     1
## 2         2     4     1     4
## 3         3     4     2     6
```

Here's the pre-Figure 9.3 data McElreath presented at the bottom of page 275.

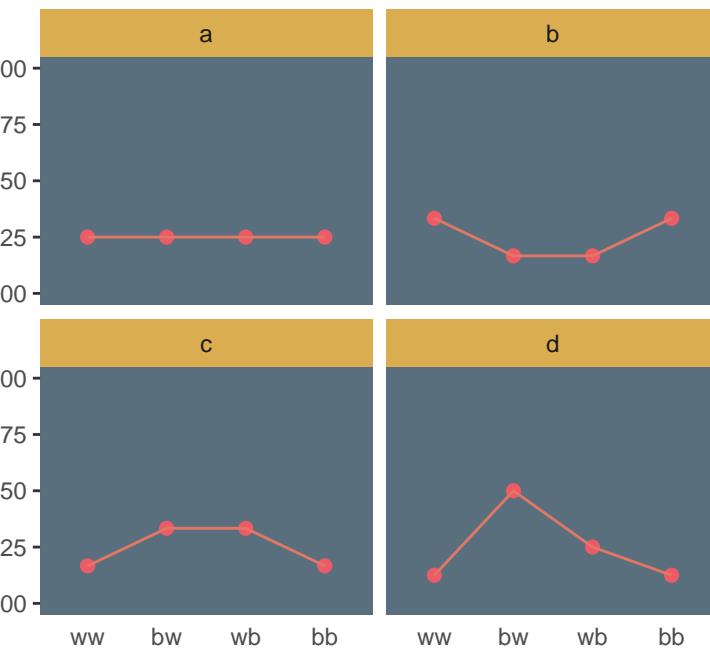
```
(  
d <-  
tibble(distribution = letters[1:4],  
       ww = c(1/4, 2/6, 1/6, 1/8),  
       bw = c(1/4, 1/6, 2/6, 4/8),  
       wb = c(1/4, 1/6, 2/6, 2/8),  
       bb = c(1/4, 2/6, 1/6, 1/8))  
)
```

```
## # A tibble: 4 x 5
##   distribution     ww     bw     wb     bb
##       <chr>     <dbl> <dbl> <dbl> <dbl>
## 1 a          0.25  0.25  0.25  0.25
## 2 b          0.333 0.167 0.167 0.333
## 3 c          0.167 0.333 0.333 0.167
## 4 d          0.125 0.5    0.25  0.125
```

Those data take just a tiny bit of wrangling before they're ready to plot with.

```
d %>%
  gather(key, value, -distribution) %>%
  mutate(key = factor(key, levels = c("ww", "bw", "wb", "bb"))) %>%

  ggplot(aes(x = key, y = value, group = 1)) +
  geom_point(size = 2, color = ghibli_palette("PonyoMedium")[4]) +
  geom_line(color = ghibli_palette("PonyoMedium")[5]) +
  coord_cartesian(ylim = 0:1) +
  labs(x = NULL,
       y = NULL) +
  theme(panel.grid    = element_blank(),
        axis.ticks.x = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("PonyoMedium")[2]),
        strip.background = element_rect(fill = ghibli_palette("PonyoMedium")[6])) +
  facet_wrap(~distribution)
```



If we go step by step, we might count the expected value for each distribution like follows.

```
d %>%
  gather(sequence, probability, -distribution) %>%
  # `str_count()` will count the number of times "b" occurs within a given row of `sequence`
  mutate(n_b      = str_count(sequence, "b")) %>%
  mutate(product = probability * n_b) %>%
  group_by(distribution) %>%
  summarise(expected_value = sum(product))

## # A tibble: 4 x 2
##   distribution expected_value
##   <chr>           <dbl>
## 1 a                 1
## 2 b                 1
## 3 c                 1
## 4 d                 1
```

We can use the same `gather()` and `group_by()` strategies on the way to computing the entropies.

```
d %>%
  gather(sequence, probability, -distribution) %>%
  group_by(distribution) %>%
  summarise(entropy = -sum(probability * log(probability)))

## # A tibble: 4 x 2
##   distribution entropy
##   <chr>           <dbl>
## 1 a                 1.39
## 2 b                 1.33
## 3 c                 1.33
## 4 d                 1.21
```

Like in the text, `distribution == "a"` had the largest `entropy` of the four. In the next example, the expected value = 1.4 and $p = .7$.

```
p <- 0.7
(
  a <-
  c((1 - p)^2,
    p * (1 - p),
    (1 - p) * p,
    p^2)
)
## [1] 0.09 0.21 0.21 0.49
```

Here's the entropy for our distribution `a`.

```
-sum(a * log(a))
```

```
## [1] 1.221729
```

I'm going to alter McElreath's simulation function from R code block 9.9 to take a seed argument. In addition, I altered the names of the objects within the function and changed the output to a tibble that will also include the conditions "ww", "bw", "wb", and "bb".

```
sim_p <- function(seed, g = 1.4) {
  set.seed(seed)

  x_123 <- runif(3)
  x_4 <- ((g * sum(x_123) - x_123[2] - x_123[3]) / (2 - g))
  z <- sum(c(x_123, x_4))
  p <- c(x_123, x_4) / z
  tibble(h = -sum(p * log(p)),
         p = p,
         key = factor(c("ww", "bw", "wb", "bb"), levels = c("ww", "bw", "wb", "bb")))
}
```

For a given `seed` and `g` value, our augmented `sim_p()` function returns a 4×3 tibble.

```
sim_p(seed = 9.9, g = 1.4)

## # A tibble: 4 x 3
##       h     p key
##   <dbl> <dbl> <fct>
## 1  1.02  0.197 ww
## 2  1.02  0.0216 bw
## 3  1.02  0.184 wb
## 4  1.02  0.597 bb
```

So the next step is to determine how many replications we'd like, create a tibble with seed values ranging from 1 to that number, and then feed those `seed` values into `sim_p()` via `purrr::map2()`, which will return a nested tibble. We'll then `unnest()` and take a peek.

```
# how many replications would you like?
n_rep <- 1e5

d <-
  tibble(seed = 1:n_rep) %>%
  mutate(sim = map2(seed, 1.4, sim_p)) %>%
  unnest()

head(d)
```

```
## # A tibble: 6 x 4
##   seed     h     p key
##   <int> <dbl> <dbl> <fct>
## 1     1  1.21  0.108  ww
## 2     1  1.21  0.151  bw
## 3     1  1.21  0.233  wb
## 4     1  1.21  0.508  bb
## 5     2  1.21  0.0674 ww
## 6     2  1.21  0.256  bw
```

In order to intelligently choose which four replications we want to highlight in Figure 9.4, we'll want to rank order them by entropy, h .

```
ranked_d <-
  d %>%
  group_by(seed) %>%
  arrange(desc(h)) %>%
  ungroup() %>%
  # here's the rank order step
  mutate(rank = rep(1:n_rep, each = 4))

head(ranked_d)
```

```
## # A tibble: 6 x 5
##   seed     h     p key   rank
##   <int> <dbl> <dbl> <fct> <int>
## 1 55665  1.22  0.0903  ww      1
## 2 55665  1.22  0.209   bw      1
## 3 55665  1.22  0.210   wb      1
## 4 55665  1.22  0.490   bb      1
## 5 71132  1.22  0.0902  ww      2
## 6 71132  1.22  0.210   bw      2
```

And we'll also want a subset of the data to correspond to McElreath's “A” through “D” distributions.

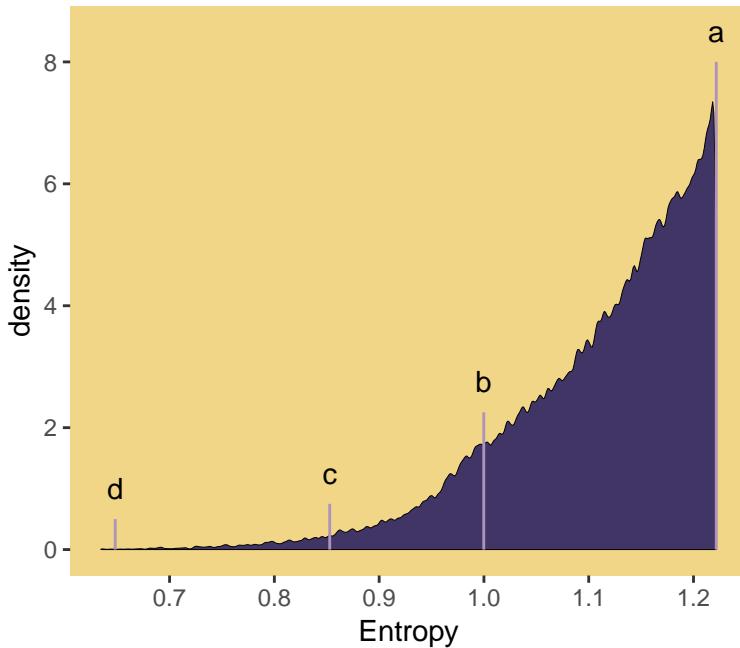
```
subset_d <-
  ranked_d %>%
  # I arrived at these `rank` values by trial and error
  filter(rank %in% c(1, 87373, n_rep - 1500, n_rep - 10)) %>%
  # I arrived at the `height` values by trial and error, too
  mutate(height      = rep(c(8, 2.25, .75, .5), each = 4),
         distribution = rep(letters[1:4], each = 4))

head(subset_d)
```

```
## # A tibble: 6 x 7
##   seed     h     p key   rank height distribution
##   <int> <dbl> <dbl> <fct> <int> <dbl> <chr>
## 1 55665  1.22  0.0903  ww      1     8     a
## 2 55665  1.22  0.209   bw      1     8     a
## 3 55665  1.22  0.210   wb      1     8     a
## 4 55665  1.22  0.490   bb      1     8     a
## 5 50981  1.000 0.0459  ww     87373  2.25  b
## 6 50981  1.000 0.0459  bw     87373  2.25  b
```

We're finally ready to plot the left panel of Figure 9.4.

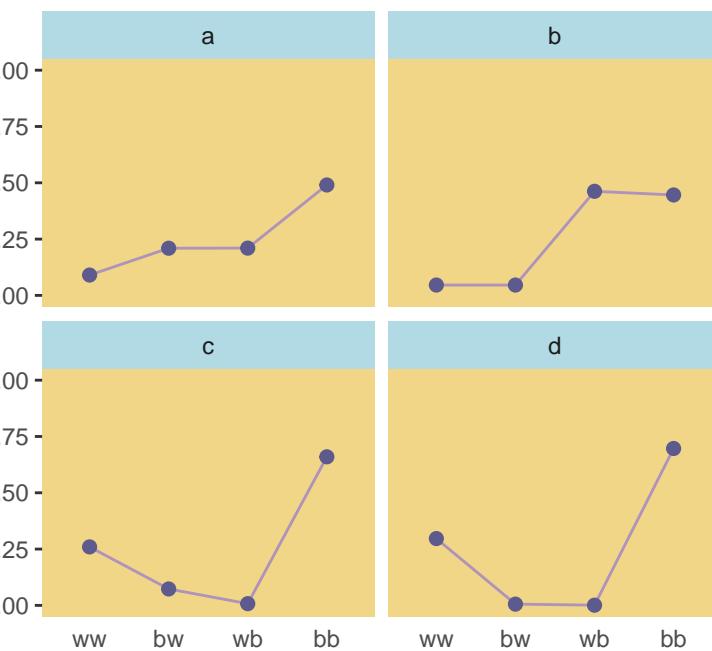
```
d %>%
  ggplot(aes(x = h)) +
  geom_density(size = 0, fill = ghibli_palette("LaputaMedium")[3],
               adjust = 1/4) +
  # note the data statements for the next two geoms
  geom_linerange(data = subset_d %>% group_by(seed) %>% slice(1),
                 aes(ymin = 0, ymax = height),
                 color = ghibli_palette("LaputaMedium")[5]) +
  geom_text(data = subset_d %>% group_by(seed) %>% slice(1),
            aes(y = height + .5, label = distribution)) +
  scale_x_continuous("Entropy",
                     breaks = seq(from = .7, to = 1.2, by = .1)) +
  theme(panel.grid      = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("LaputaMedium")[7]))
```



Did you notice how our `adjust = 1/4` with `geom_density()` served a similar function to the `adj=0.1` in McElreath's `dens()` code. Anyways, here's the right panel.

```
ranked_d %>%
  filter(rank %in% c(1, 87373, n_rep - 1500, n_rep - 10)) %>%
  mutate(distribution = rep(letters[1:4], each = 4)) %>%

  ggplot(aes(x = key, y = p, group = 1)) +
  geom_line(color = ghibli_palette("LaputaMedium")[5]) +
  geom_point(size = 2, color = ghibli_palette("LaputaMedium")[4]) +
  coord_cartesian(ylim = 0:1) +
  labs(x = NULL,
       y = NULL) +
  theme(panel.grid      = element_blank(),
        axis.ticks.x = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("LaputaMedium")[7]),
        strip.background = element_rect(fill = ghibli_palette("LaputaMedium")[6])) +
  facet_wrap(~distribution)
```



Because we were simulating, our values won't match up identically with those in the text. But we're pretty close, eh?

Since we saved our `sim_p()` output in a nested tibble, which we then `unnested()`, there's no need to separate the entropy values from the distributional values the way McElreath did in R code 9.11. If we wanted to determine our highest entropy value—and the corresponding `seed` and `p` values, while we're at it—, we might use `max(h)` within `slice()`.

```
ranked_d %>%
  group_by(key) %>%
  slice(max(h))
```

```
## # A tibble: 4 x 5
## # Groups:   key [4]
##   seed     h     p key    rank
##   <int> <dbl> <dbl> <fct> <int>
## 1 55665  1.22  0.0903 ww      1
## 2 55665  1.22  0.209  bw      1
## 3 55665  1.22  0.210  wb      1
## 4 55665  1.22  0.490  bb      1
```

That maximum `h` value matched up nicely with the one in the text. If you look at the `p` column, you'll see our values approximated McElreath's `distribution` values, too. In both cases, they're real close to the `a` values we computed, above.

a

```
## [1] 0.09 0.21 0.21 0.49
```

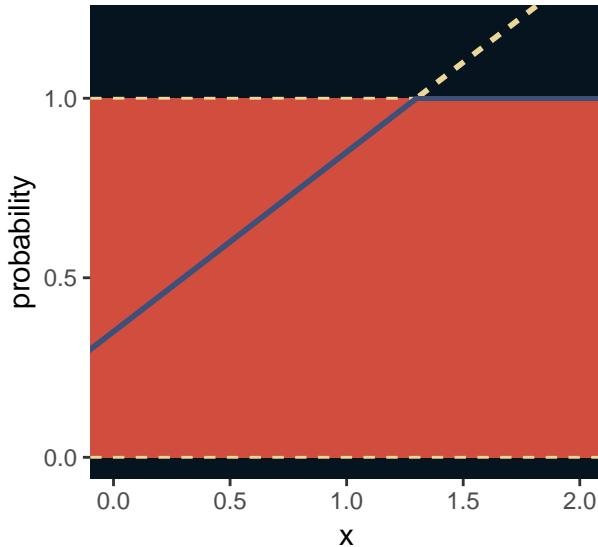
9.2 Generalized linear models

For an outcome variable that is continuous and far from any theoretical maximum or minimum, [a simple] Gaussian model has maximum entropy. But when the outcome variable is either discrete or bounded, a Gaussian likelihood is not the most powerful choice. (p. 280)

I winged the values for our Figure 9.5.

```
tibble(x = seq(from = -1, to = 3, by = .01)) %>%
  mutate(probability = .35 + x * .5) %>%
```

```
ggplot(aes(x = x, y = probability)) +
  geom_rect(xmin = -1, xmax = 3,
            ymin = 0, ymax = 1,
            fill = ghibli_palette("MononokeMedium")[5]) +
  geom_hline(yintercept = 0:1, linetype = 2, color = ghibli_palette("MononokeMedium")[7]) +
  geom_line(aes(linetype = probability > 1, color = probability > 1),
            size = 1) +
  geom_segment(x = 1.3, xend = 3,
               y = 1, yend = 1,
               size = 2/3, color = ghibli_palette("MononokeMedium")[3]) +
  scale_color_manual(values = c(ghibli_palette("MononokeMedium")[3],
                               ghibli_palette("MononokeMedium")[7])) +
  scale_y_continuous(breaks = c(0, .5, 1)) +
  coord_cartesian(xlim = 0:2,
                  ylim = c(0, 1.2)) +
  theme(panel.grid      = element_blank(),
        legend.position = "none",
        panel.background = element_rect(fill = ghibli_palette("MononokeMedium")[1]))
```



For a count outcome y for which each observation arises from n trials and with constant expected value np , the binomial distribution has maximum entropy. So it's the least informative distribution that satisfies our prior knowledge of the outcomes y . (p. 281)

The binomial model follows the basic form

$$\begin{aligned} y_i &\sim \text{Binomial}(n, p_i) \\ f(p_i) &= \alpha + \beta x_i \end{aligned}$$

The $f()$ portion of the second line represents the link function. We need the link function because, though the shape of the Binomial distribution is determined by two parameters— n and p —, neither is equivalent to the Gaussian mean μ . The mean outcome, rather, is np —a function of both. The link function also ensures the model doesn't make probability predictions outside of the boundary $[0, 1]$.

Let's get more general.

9.2.1 Meet the family.

Here are the Gamma and Exponential panels for Figure 9.6.

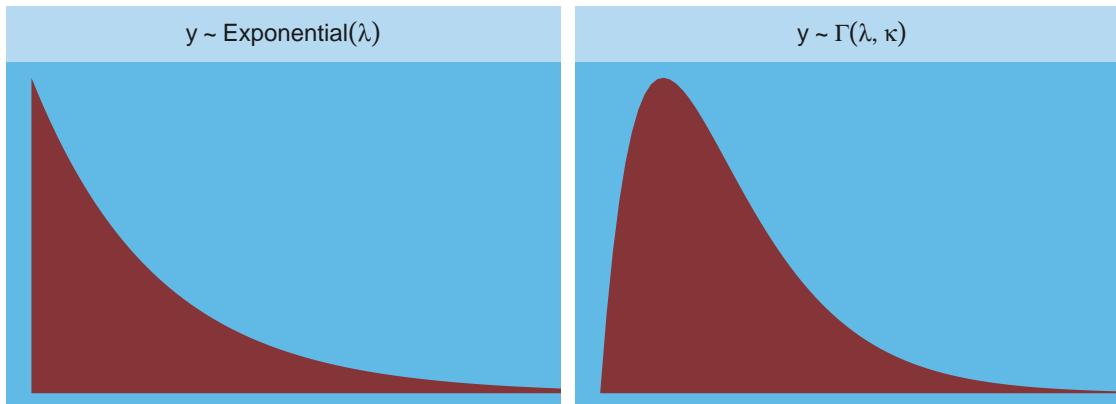
```

length_out <- 100

tibble(x = seq(from = 0, to = 5, length.out = length_out)) %>%
  mutate(Gamma      = dgamma(x, 2, 2),
         Exponential = dexp(x)) %>%
gather(key, density, -x) %>%
mutate(label = rep(c("y ~ Gamma(lambda, kappa)", "y ~ Exponential(lambda)"), each = n()/2)) %>%

ggplot(aes(x = x, ymin = 0, ymax = density)) +
  geom_ribbon(fill = ghibli_palette("SpiritedMedium")[3]) +
  scale_x_continuous(NULL, breaks = NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:4) +
  theme(panel.grid      = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("SpiritedMedium")[5]),
        strip.background = element_rect(fill = ghibli_palette("SpiritedMedium")[7])) +
  facet_wrap(~label, scales = "free_y", labeller = label_parsed)

```



The Gaussian:

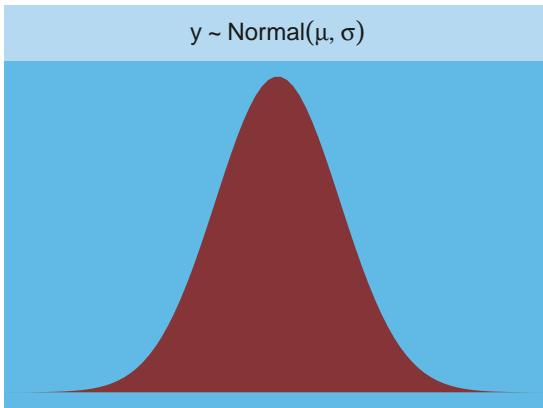
```

length_out <- 100

tibble(x = seq(from = -5, to = 5, length.out = length_out)) %>%
  mutate(density = dnorm(x),
         strip   = "y ~ Normal(mu, sigma)") %>%

ggplot(aes(x = x, ymin = 0, ymax = density)) +
  geom_ribbon(fill = ghibli_palette("SpiritedMedium")[3]) +
  scale_x_continuous(NULL, breaks = NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = -4:4) +
  theme(panel.grid      = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("SpiritedMedium")[5]),
        strip.background = element_rect(fill = ghibli_palette("SpiritedMedium")[7])) +
  facet_wrap(~strip, labeller = label_parsed)

```

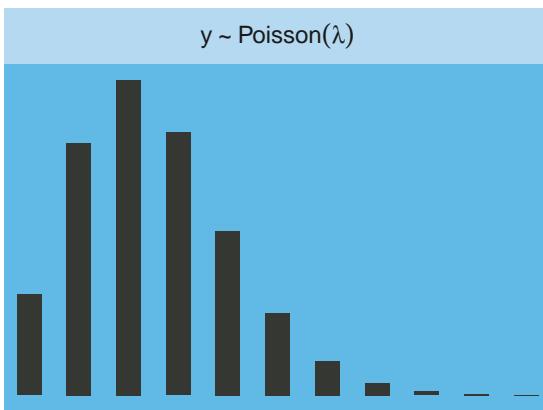


Here is the Poisson.

```
length_out <- 100

tibble(x = 0:20) %>%
  mutate(density = dpois(x, lambda = 2.5),
        strip    = "y %~% Poisson(lambda)") %>%

  ggplot(aes(x = x, y = density)) +
  geom_col(fill = ghibli_palette("SpiritedMedium")[2], width = 1/2) +
  scale_x_continuous(NULL, breaks = NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:10) +
  theme(panel.grid      = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("SpiritedMedium")[5]),
        strip.background = element_rect(fill = ghibli_palette("SpiritedMedium")[7])) +
  facet_wrap(~strip, labeller = label_parsed)
```



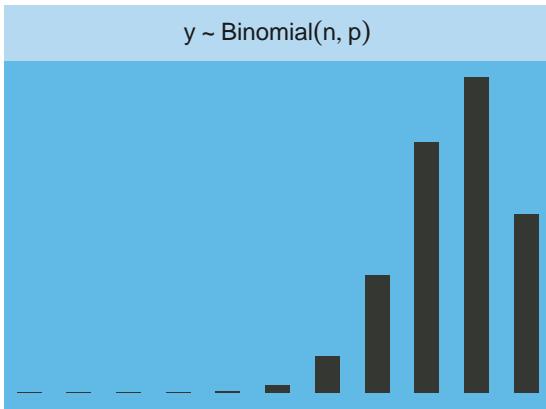
Finally, the Binomial:

```
length_out <- 100

tibble(x = 0:10) %>%
  mutate(density = dbinom(x, size = 10, prob = .85),
        strip    = "y %~% Binomial(n, p)") %>%

  ggplot(aes(x = x, y = density)) +
  geom_col(fill = ghibli_palette("SpiritedMedium")[2], width = 1/2) +
  scale_x_continuous(NULL, breaks = NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:10) +
  theme(panel.grid      = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("SpiritedMedium")[5]),
```

```
strip.background = element_rect(fill = ghibli_palette("SpiritedMedium")[7])) +
facet_wrap(~strip, labeller = label_parsed)
```



9.2.1.1 Rethinking: A likelihood is a prior.

In traditional statistics, likelihood functions are “objective” and prior distributions “subjective.” However, likelihoods are themselves prior probability distributions: They are priors for the data, conditional on the parameters. And just like with other priors, there is no correct likelihood. But there are better and worse likelihoods, depending upon context. (p. 284)

For a little more in this, check out McElreath’s great lecture, [Bayesian Statistics without Frequentist Language](#). This subsection also reminds me of the title of one of Gelman’s blog posts, *“It is perhaps merely an accident of history that skeptics and subjectivists alike strain on the gnat of the prior distribution while swallowing the camel that is the likelihood”*. The title, which itself is a quote, comes from one of his papers, which he linked to in the blog, along with several related papers. It’s taken some time for the weight of that quote to sink in with me, and indeed it’s still sinking. Perhaps you’ll benefit from it, too.

9.2.2 Linking linear models to distributions.

To build a regression model from any of the exponential family distributions is just a matter of attaching one or more linear models to one or more of the parameters that describe the distribution’s shape. But as hinted at earlier, usually we require a link function to prevent mathematical accidents like negative distances or probability masses that exceed 1. (p. 284)

These models generally follow the form

$$\begin{aligned} y_i &\sim \text{Some distribution}(\theta_i, \phi) \\ f(\theta_i) &= \alpha + \beta x_i \end{aligned}$$

where θ_i is a parameter of central interest (e.g., the probability of 1 in a Binomial distribution) and ϕ is a placeholder for any other parameters necessary for the likelihood but not of primary substantive interest (e.g., σ in work-a-day Gaussian models). And as stated earlier, $f()$ is the link function.

Speaking of,

the logit link maps a parameter that is defined as a probability mass and therefore constrained to lie between zero and one, onto a linear model that can take on any real value. This link is extremely common when working with binomial GLMs. In the context of a model definition, it looks like this:

$$\begin{aligned} y_i &\sim \text{Binomial}(n, p_i) \\ \text{logit}(p_i) &= \alpha + \beta x_i \end{aligned}$$

And the logit function itself is defined as the *log-odds*:

$$\text{logit}(p_i) = \log \frac{p_i}{1 - p_i}$$

The “odds” of an event are just the probability it happens divided by the probability it does not happen. So really all that is being stated here is:

$$\log \frac{p_i}{1 - p_i} = \alpha + \beta x_i$$

If we do the final algebraic manipulation on page 285, we can solve for p_i in terms of the linear model:

$$p_i = \frac{\exp(\alpha + \beta x_i)}{1 + \exp(\alpha + \beta x_i)}$$

As we'll see later, we will make great use of this formula via the `brms::inv_logit_scaled()` when making sense of logistic regression models. Now we have that last formula in hand, we can make the data necessary for Figure 9.7.

```
# first, we'll make data for the horizontal lines
alpha <- 0
beta  <- 4

lines <-
  tibble(x           = seq(from = -1, to = 1, by = .25)) %>%
  mutate(`log-odds` = alpha + x * beta,
        probability = exp(alpha + x * beta) / (1 + exp(alpha + x * beta)))

# now we're ready to make the primary data
beta  <- 2

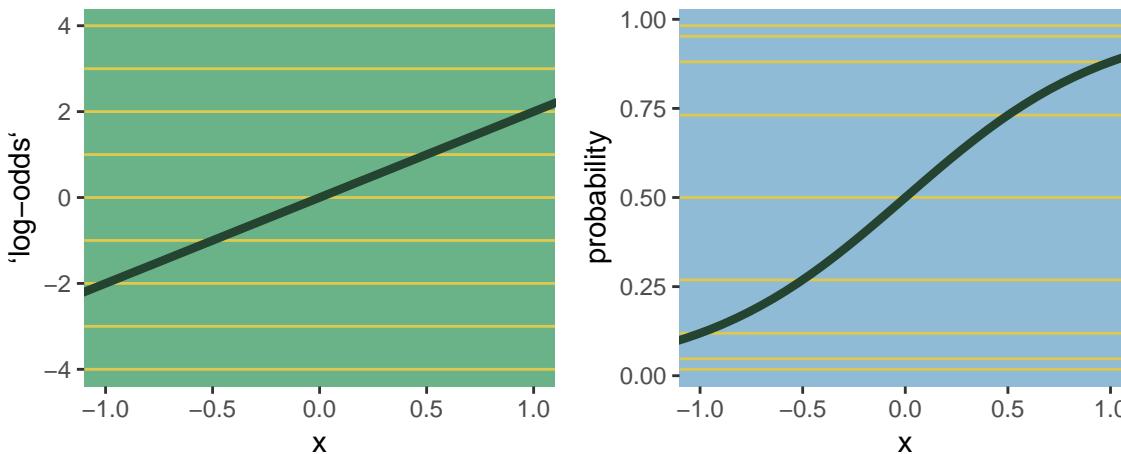
d <-
  tibble(x           = seq(from = -1.5, to = 1.5, length.out = 50)) %>%
  mutate(`log-odds` = alpha + x * beta,
        probability = exp(alpha + x * beta) / (1 + exp(alpha + x * beta)))

# now we make the individual plots
p1 <-
  d %>%
  ggplot(aes(x = x, y = `log-odds`)) +
  geom_hline(data = lines,
             aes(yintercept = `log-odds`),
             color = ghibli_palette("YesterdayMedium")[6]) +
  geom_line(size = 1.5, color = ghibli_palette("YesterdayMedium")[3]) +
  coord_cartesian(xlim = -1:1) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("YesterdayMedium")[5]))

p2 <-
  d %>%
  ggplot(aes(x = x, y = probability)) +
  geom_hline(data = lines,
             aes(yintercept = probability),
             color = ghibli_palette("YesterdayMedium")[6]) +
  geom_line(size = 1.5, color = ghibli_palette("YesterdayMedium")[3]) +
  coord_cartesian(xlim = -1:1) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("YesterdayMedium")[7]))

# finally, we're ready to mash the plots together and behold their nerdy glory
```

```
library(gridExtra)
grid.arrange(p1, p2, ncol = 2)
```



The key lesson for now is just that no regression coefficient, such as β , from a GLM ever produces a constant change on the outcome scale. Recall that we defined interaction (Chapter 7) as a situation in which the effect of a predictor depends upon the value of another predictor. Well now every predictor essentially interacts with itself, because the impact of a change in a predictor depends upon the value of the predictor before the change...

The second very common link function is the log link. This link function maps a parameter that is defined over only positive real values onto a linear model. For example, suppose we want to model the standard deviation of σ of a Gaussian distribution so it is a function of a predictor variable x . The parameter σ must be positive, because a standard deviation cannot be negative nor can it be zero. The model might look like:

$$y_i \sim \text{Normal}(\mu, \sigma_i)$$

$$\log(\sigma_i) = \alpha + \beta x_i$$

In this model, the mean μ is constant, but the standard deviation scales with the value x_i . (p. 268)

This kind of model is trivial in the brms framework, which you can learn more about in Bürkner's vignette [Estimating Distributional Models with brms](#). Before moving on with the text, let's detour and see how we might fit such a model. First, let's simulate some continuous data y for which the SD is effected by a dummy variable x .

```
set.seed(9)
(
  d <-
  tibble(x = rep(0:1, each = 100)) %>%
  mutate(y = rnorm(n = n(), mean = 100, sd = 10 + x * 10))
)
```

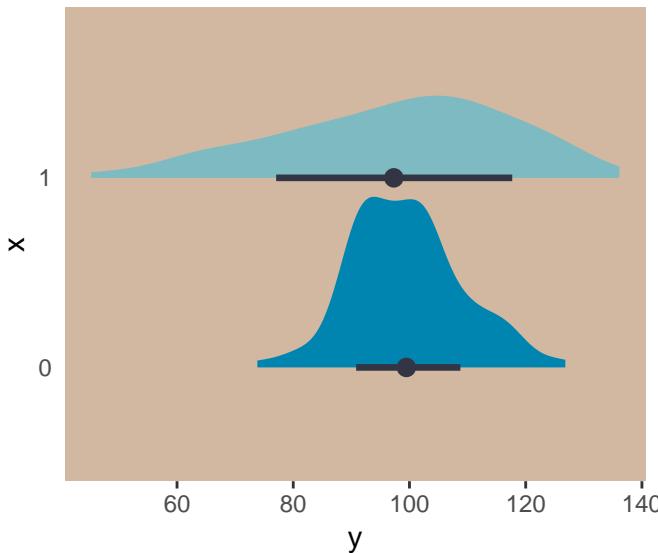
```
## # A tibble: 200 x 2
##       x     y
##   <int> <dbl>
## 1     0  92.3
## 2     0  91.8
## 3     0  98.6
## 4     0  97.2
## 5     0 104.
## 6     0  88.1
## 7     0 112.
## 8     0  99.8
## 9     0  97.5
## 10    0  96.4
## # ... with 190 more rows
```

We can view what data like these look like with aid from `tidybayes::geom_halfeyeh()`.

```
library(tidybayes)

d %>%
  mutate(x = x %>% as.character()) %>%

  ggplot(aes(x = y, y = x, fill = x)) +
  geom_halfeyeh(color = ghibli_palette("KikiMedium")[2],
                 point_interval = mean_qi, .width = .68) +
  scale_fill_manual(values = c(ghibli_palette("KikiMedium")[4],
                               ghibli_palette("KikiMedium")[6])) +
  theme(panel.grid      = element_blank(),
        axis.ticks.y = element_blank(),
        legend.position = "none",
        panel.background = element_rect(fill = ghibli_palette("KikiMedium")[7]))
```



Even though the means of y are the same for both levels of the x dummy, the variance for $x == 1$ is substantially larger than that for $x == 0$. Let's open brms.

```
library(brms)
```

For such a model, we have two formulas: one for μ and one for σ . We wrap both within the `bf()` function.

```
b9.1 <-
  brm(data = d,
       family = gaussian,
       bf(y ~ 1, sigma ~ 1 + x),
       prior = c(prior(normal(100, 10), class = Intercept),
                 prior(normal(0, 10), class = Intercept, dpar = sigma),
                 prior(normal(0, 10), class = b, dpar = sigma)),
       seed = 9)
```

Do note our use of the `dpar` arguments in the `prior` statements. Here's the summary.

```
print(b9.1)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = log
## Formula: y ~ 1
```

```

##           sigma ~ 1 + x
## Data: d (Number of observations: 200)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      99.04     0.86    97.37   100.77      4170 1.00
## sigma_Intercept  2.26     0.07     2.13     2.40      3784 1.00
## sigma_x         0.73     0.10     0.53     0.92      4257 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Now we get an intercept for both μ and σ , with the intercept for sigma identified as `sigma_Intercept`. And note the coefficient for σ was named `sigma_x`. Also notice the scale the `sigma_` coefficients are on. These are not in the original metric, but rather based on `log()`. You can confirm that by the second line of the `print()` output: `Links: mu = identity; sigma = log`. So if you want to get a sense of the effects of `x` on the σ for `y`, you have to exponentiate the formula. Here we'll do so with the `posterior_samples()`.

```

post <- posterior_samples(b9.1)

head(post)

```

```

##   b_Intercept b_sigma_Intercept b_sigma_x      lp__
## 1 99.75865      2.281731 0.6406836 -818.1893
## 2 99.83593      2.244206 0.7046929 -817.9380
## 3 98.40659      2.282739 0.7512565 -817.9899
## 4 98.59265      2.222575 0.6913024 -818.1270
## 5 98.05085      2.244358 0.7482665 -818.1277
## 6 99.71570      2.306853 0.6402182 -818.0676

```

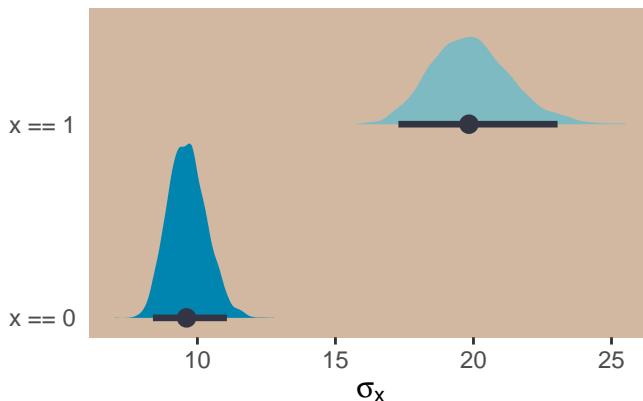
With the samples in hand, we'll use the model formula to compute the model-implied standard deviations of `y` based on the `x` dummy and then examine them in a plot.

```

post %>%
  transmute(`x == 0` = exp(b_sigma_Intercept + b_sigma_x * 0),
            `x == 1` = exp(b_sigma_Intercept + b_sigma_x * 1)) %>%
  gather(key, sd) %>%
  ggplot(aes(x = sd, y = key, fill = key)) +
  geom_halfeyeh(color = ghibli_palette("KikiMedium")[2],
                 point_interval = median_qi, .width = .95) +
  scale_fill_manual(values = c(ghibli_palette("KikiMedium")[4],
                             ghibli_palette("KikiMedium")[6])) +
  labs(x = expression(sigma[x]), y = NULL,
       subtitle = expression(paste("Model-implied ", italic(SD), "s by group x")))) +
  coord_cartesian(ylim = c(1.5, 2)) +
  theme(panel.grid      = element_blank(),
        axis.ticks.y = element_blank(),
        legend.position = "none",
        panel.background = element_rect(fill = ghibli_palette("KikiMedium")[7]))

```

Model-implied SDs by group x



And if we looked back at the data, those *SD* estimates are just what we'd expect.

```
d %>%
  group_by(x) %>%
  summarise(sd = sd(y) %>% round(digits = 1))
```

```
## # A tibble: 2 x 2
##       x     sd
##   <int> <dbl>
## 1     0    9.6
## 2     1   19.8
```

For more on models like this, check out Christakis' [2014: What scientific idea is ready for retirement?](#) or [The “average” treatment effect: A construct ripe for retirement. A commentary on Deaton and Cartwright](#). Kruschke also covered modeling σ a bit in his [Doing Bayesian Data Analysis, Second Edition: A Tutorial with R, JAGS, and Stan](#). Finally, this is foreshadowing a bit because it requires the multilevel model (see Chapters 12 and 13), but you might also check out the preprint by Williams, Liu, Martin, and Rast, [Bayesian Multivariate Mixed-Effects Location Scale Modeling of Longitudinal Relations among Affective Traits, States, and Physical Activity](#).

But getting back to the text,

what the log link effectively assumes is that the parameter's value is the exponentiation of the linear model. Solving $\log(\sigma_i) = \alpha + \beta x_i$ for σ_i yields the inverse link:

$$\sigma_i = \exp(\alpha + \beta x_i)$$

The impact of this assumption can be seen in [our version of] Figure 9.8. (pp. 286—287)

```
# first, we'll make data that'll be make the horizontal lines
alpha <- 0
beta  <- 2

lines <-
  tibble(`log-measurement` = -3:3) %>%
  mutate(`original measurement` = exp(`log-measurement`))

# now we're ready to make the primary data
d <-
  tibble(x           = seq(from = -1.5, to = 1.5, length.out = 50)) %>%
  mutate(`log-measurement` = alpha + x * beta,
        `original measurement` = exp(alpha + x * beta))

# now we make the individual plots
p1 <-
```

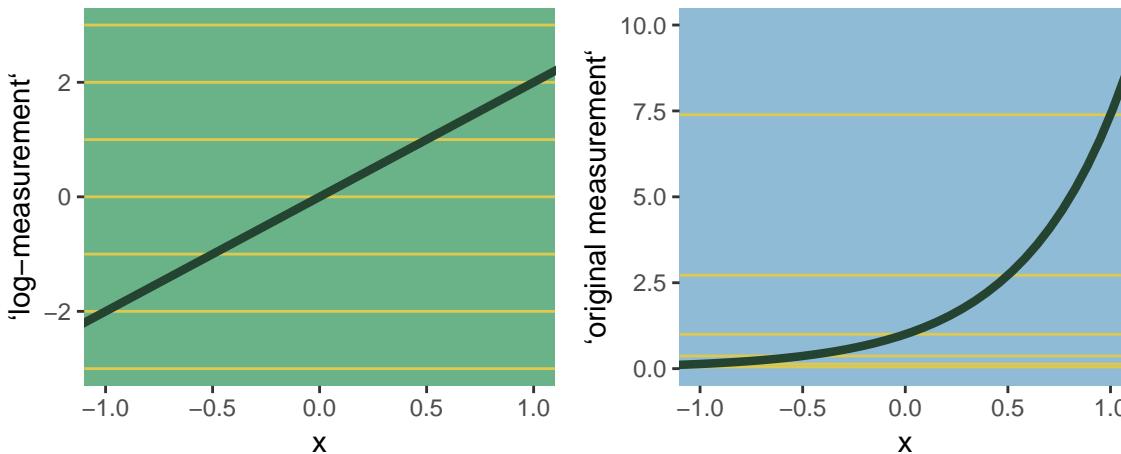
```

d %>%
  ggplot(aes(x = x, y = `log-measurement`)) +
  geom_hline(data = lines,
             aes(yintercept = `log-measurement`),
             color = ghibli_palette("YesterdayMedium")[6]) +
  geom_line(size = 1.5, color = ghibli_palette("YesterdayMedium")[3]) +
  coord_cartesian(xlim = -1:1) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("YesterdayMedium")[5]))

p2 <-
d %>%
  ggplot(aes(x = x, y = `original measurement`)) +
  geom_hline(data = lines,
             aes(yintercept = `original measurement`),
             color = ghibli_palette("YesterdayMedium")[6]) +
  geom_line(size = 1.5, color = ghibli_palette("YesterdayMedium")[3]) +
  coord_cartesian(xlim = -1:1,
                  ylim = 0:10) +
  theme(panel.grid = element_blank(),
        panel.background = element_rect(fill = ghibli_palette("YesterdayMedium")[7]))

# finally, we're ready to mash the plots together and behold their nerdy glory
grid.arrange(p1, p2, ncol = 2)

```



Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```

sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib

```

```
##  
## locale:  
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8  
##  
## attached base packages:  
## [1] stats      graphics   grDevices  utils      datasets   methods    base  
##  
## other attached packages:  
## [1] brms_2.8.8       Rcpp_1.0.1       tidybayes_1.0.4  gridExtra_2.3  
## [5] ghibli_0.2.0    forcats_0.3.0    stringr_1.4.0    dplyr_0.8.0.1  
## [9] purrrr_0.2.5    readr_1.1.1     tidyrr_0.8.1    tibble_2.1.1  
## [13] ggplot2_3.1.1   tidyverse_1.2.1  
##  
## loaded via a namespace (and not attached):  
## [1] nlme_3.1-137          matrixStats_0.54.0  
## [3] xts_0.10-2           lubridate_1.7.4  
## [5] threejs_0.3.1         httr_1.3.1  
## [7] rstan_2.18.2          rprojroot_1.3-2  
## [9] tools_3.5.1           backports_1.1.4  
## [11] utf8_1.1.4            R6_2.3.0  
## [13] DT_0.4                lazyeval_0.2.2  
## [15] colorspace_1.3-2     withr_2.1.2  
## [17] prettyunits_1.0.2    processx_3.2.1  
## [19] tidyselect_0.2.5     Brobdingnag_1.2-6  
## [21] compiler_3.5.1      cli_1.0.1  
## [23] rvest_0.3.2          arrayhelpers_1.0-20160527  
## [25] shinyjs_1.0           xml2_1.2.0  
## [27] labeling_0.3          colourpicker_1.0  
## [29] bookdown_0.9          scales_1.0.0  
## [31] dygraphs_1.1.1.5     mvtnorm_1.0-10  
## [33] callr_3.1.0          ggridges_0.5.0  
## [35] StanHeaders_2.18.0-1 digest_0.6.18  
## [37] rmarkdown_1.10         base64enc_0.1-3  
## [39] pkgconfig_2.0.2       htmltools_0.3.6  
## [41] htmlwidgets_1.2        rlang_0.3.4  
## [43] readxl_1.1.0          rstudioapi_0.7  
## [45] shiny_1.1.0            generics_0.0.2  
## [47] svUnit_0.7-12         zoo_1.8-2  
## [49] jsonlite_1.5           gtools_3.8.1  
## [51] crosstalk_1.0.0       inline_0.3.15  
## [53] magrittr_1.5            loo_2.1.0  
## [55] bayesplot_1.6.0       Matrix_1.2-14  
## [57] munsell_0.5.0          fansi_0.4.0  
## [59] abind_1.4-5            stringi_1.4.3  
## [61] yaml_2.1.19           pkgbuild_1.0.2  
## [63] plyr_1.8.4              ggstance_0.3  
## [65] grid_3.5.1              parallel_3.5.1  
## [67] promises_1.0.1          crayon_1.3.4  
## [69] miniUI_0.1.1.1         lattice_0.20-35  
## [71] haven_1.1.2             hms_0.4.2  
## [73] ps_1.2.1                knitr_1.20  
## [75] pillar_1.3.1            igraph_1.2.1  
## [77] markdown_0.8             shinystan_2.5.0  
## [79] stats4_3.5.1            reshape2_1.4.3  
## [81] rstantools_1.5.1        glue_1.3.1.9000  
## [83] evaluate_0.10.1          modelr_0.1.2  
## [85] httpuv_1.4.4.2          cellranger_1.1.0  
## [87] gtable_0.3.0             assertthat_0.2.0  
## [89] xfun_0.3                 mime_0.5  
## [91] xtable_1.8-2            broom_0.5.1
```

```
## [93] coda_0.19-2          later_0.7.3
## [95] rsconnect_0.8.8        shinythemes_1.1.1
## [97] bridgesampling_0.6-0
```

Chapter 10

Counting and Classification

All over the world, every day, scientists throw away information. Sometimes this is through the removal of “outliers,” cases in the data that offend the model and are exiled. More routinely, counted things are converted to proportions before analysis. Why does analysis of proportions throw away information? Because $10/20$ and $\frac{1}{2}$ are the same proportion, one-half, but have very different sample sizes. Once converted to proportions, and treated as outcomes in a linear regression, the information about sample size has been destroyed.

It’s easy to retain the information about sample size. All that is needed is to model what has actually been observed, the counts instead of the proportions. (p. 291)

In this chapter, we focus on the two most common types of count models: the binomial and the Poisson.

Side note: For a nice Bayesian way to accommodate outliers in your Gaussian models, check out my blog [Robust Linear Regression with Student's t-Distribution](#).

10.1 Binomial regression

The basic binomial model follows the form

$$y \sim \text{Binomial}(n, p)$$

where y is some count variable, n is the number of trials, and p it the probability a given trial was a 1, which is sometimes termed a *success*. When $n = 1$, then y is a vector of 0s and 1s. Presuming the logit link, models of this type are commonly termed logistic regression. When $n > 1$, and still presuming the logit link, we might call our model an aggregated logistic regression model, or more generally an aggregated binomial regression model.

10.1.1 Logistic regression: Prosocial chimpanzees.

Load the `chimpanzees` data.

```
library(rethinking)
data(chimpanzees)
d <- chimpanzees
```

Switch from `rethinking` to `brms`.

```
detach(package:rethinking, unload = T)
library(brms)
rm(chimpanzees)
```

We start with the simple intercept-only logistic regression model, which follows the statistical formula

$$\begin{aligned}\text{pulled_left}_i &\sim \text{Binomial}(1, p_i) \\ \text{logit}(p_i) &= \alpha \\ \alpha &\sim \text{Normal}(0, 10)\end{aligned}$$

In the `brm()` formula syntax, including a | bar on the left side of a formula indicates we have extra supplementary information about our criterion. In this case, that information is that each `pulled_left` value corresponds to a single trial (i.e., `trials(1)`), which itself corresponds to the $n = 1$ portion of the statistical formula, above.

```
b10.1 <-  
  brm(data = d, family = binomial,  
        pulled_left | trials(1) ~ 1,  
        prior(normal(0, 10), class = Intercept),  
        seed = 10)
```

You might use `fixef()` to get a focused summary of the intercept.

```
library(tidyverse)  
  
fixef(b10.1) %>%  
  round(digits = 2)  
  
##           Estimate Est.Error Q2.5 Q97.5  
## Intercept     0.32      0.09 0.14   0.5
```

The `brms::inv_logit_scaled()` function will be our alternative to the `logistic()` function in rethinking.

```
c(.18, .46) %>%  
  inv_logit_scaled()  
  
## [1] 0.5448789 0.6130142  
  
fixef(b10.1) %>%  
  inv_logit_scaled()  
  
##           Estimate Est.Error      Q2.5      Q97.5  
## Intercept 0.579012 0.5226864 0.5350585 0.6217093
```

With the next two chimp models, we add predictors in the usual way.

```
b10.2 <-  
  brm(data = d, family = binomial,  
        pulled_left | trials(1) ~ 1 + prosoc_left,  
        prior = c(prior(normal(0, 10), class = Intercept),  
                  prior(normal(0, 10), class = b)),  
        seed = 10)  
  
b10.3 <-  
  update(b10.2,  
        newdata = d,  
        formula = pulled_left | trials(1) ~ 1 + prosoc_left + condition:prosoc_left)
```

Compute the WAIC for each model and save the results within the `brmfit` objects.

```
b10.1 <- add_criterion(b10.1, "waic")
b10.2 <- add_criterion(b10.2, "waic")
b10.3 <- add_criterion(b10.3, "waic")
```

Compare them with the `loo_compare()` and make sure to add the `criterion = "waic"` argument.

```
w <- loo_compare(b10.1, b10.2, b10.3, criterion = "waic")
print(w, simplify = F)
```

	elpd_diff	se_diff	elpd_waic	se_elpd_waic	p_waic	se_p_waic	waic	se_waic
## b10.2	0.0	0.0	-340.2	4.7	2.0	0.0	680.4	9.4
## b10.3	-1.0	0.4	-341.2	4.7	3.0	0.1	682.4	9.4
## b10.1	-3.8	3.1	-344.0	3.5	1.0	0.0	688.0	7.1

Recall our `cbind()` trick to convert the differences from the elpd metric to the WAIC metric.

```
cbind(waic_diff = w[, 1] * -2,
      se        = w[, 2] * 2) %>%
  round(digits = 2)
```

	waic_diff	se
## b10.2	0.00	0.00
## b10.3	2.03	0.72
## b10.1	7.57	6.21

For this chapter, we'll take our color scheme from the `wesanderson` package's Moonrise2 palette.

```
# install.packages("wesanderson", dependencies = T)
library(wesanderson)

wes_palette("Moonrise2")
```



```
wes_palette("Moonrise2")[1:4]
```

```
## [1] "#798E87" "#C27D38" "#CCC591" "#29211F"
```

We'll also take a few formatting cues from [Edward Tufte](#), courtesy of the `ggthemes` package. The `theme_tufte()` function will change the default font and remove some chart junk. The `theme_set()` function, below, will make these adjustments the default for all subsequent ggplot2 plots. To undo this, just execute `theme_set(theme_default())`.

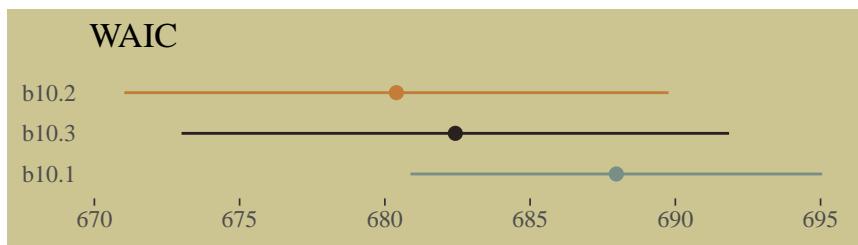
```
library(ggthemes)
library(bayesplot)

theme_set(theme_default() +
  theme_tufte() +
  theme(plot.background = element_rect(fill = wes_palette("Moonrise2")[3],
                                         color = wes_palette("Moonrise2")[3])))
```

Finally, here's our WAIC plot.

```
w %>%
  data.frame() %>%
  rownames_to_column(var = "model") %>%

  ggplot() +
  geom_pointrange(aes(x = reorder(model, -waic), y = waic,
                       ymin = waic - se_waic,
                       ymax = waic + se_waic,
                       color = model),
                   shape = 16) +
  scale_color_manual(values = wes_palette("Moonrise2")[c(1:2, 4)]) +
  coord_flip() +
  labs(x = NULL, y = NULL,
       title = "WAIC") +
  theme(axis.ticks.y = element_blank(),
        legend.position = "none")
```



The full model, b10.3, did not have the lowest WAIC value. Though note how wide those standard error bars are relative to the point estimates. There's a lot of model uncertainty there. Here are the WAIC weights.

```
model_weights(b10.1, b10.2, b10.3,
               weights = "waic")
```

```
##      b10.1      b10.2      b10.3
## 0.01638298 0.72182631 0.26179071
```

Let's look at the parameter summaries for the theory-based model.

```
print(b10.3)

## Family: binomial
## Links: mu = logit
## Formula: pulled_left | trials(1) ~ prosoc_left + prosoc_left:condition
## Data: d (Number of observations: 504)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##                               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  0.05     0.13   -0.19    0.30      3694  1.00
## prosoc_left                 0.60     0.23    0.18    1.05      2503  1.00
## prosoc_left:condition     -0.09     0.27   -0.62    0.45      2718  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Here's what the odds are multiplied by:

```
fixef(b10.3)[2] %>%
  exp()
```

```
## [1] 1.831113
```

Given an estimated value of 4, the probability of a pull, all else equal, would be close to 1.

```
inv_logit_scaled(4)
```

```
## [1] 0.9820138
```

Adding the coefficient, `fixef(b10.3)[2]`, would yield an even higher estimate.

```
(4 + fixef(b10.3)[2]) %>%
  inv_logit_scaled()
```

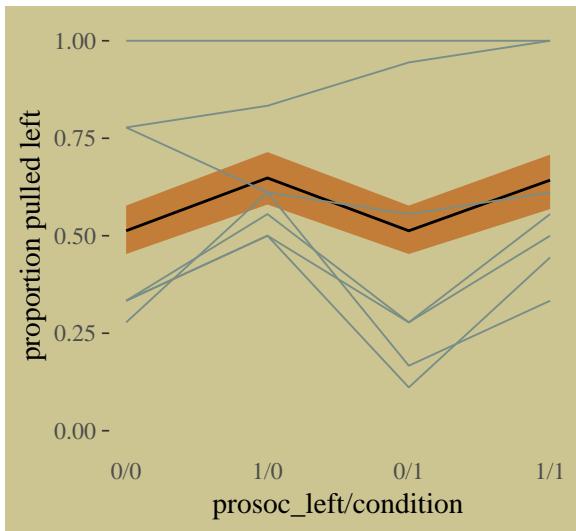
```
## [1] 0.9900966
```

For our variant of Figure 10.2, we use `brms::pp_average()` in place of `rethinking::ensemble()`.

```
# the combined `fitted()` results of the three models weighted by their WAICs
ppa <-
  pp_average(b10.1, b10.2, b10.3,
              weights = "waic",
              method = "fitted") %>%
  as_tibble() %>%
  bind_cols(b10.3$data) %>%
  distinct(Estimate, Q2.5, Q97.5, condition, prosoc_left) %>%
  mutate(x_axis = str_c(prosoc_left, condition, sep = "/")) %>%
  mutate(x_axis = factor(x_axis, levels = c("0/0", "1/0", "0/1", "1/1"))) %>%
  rename(pulled_left = Estimate)

# the empirically-based summaries
d_plot <-
  d %>%
  group_by(actor, condition, prosoc_left) %>%
  summarise(pulled_left = mean(pulled_left)) %>%
  mutate(x_axis = str_c(prosoc_left, condition, sep = "/")) %>%
  mutate(x_axis = factor(x_axis, levels = c("0/0", "1/0", "0/1", "1/1")))

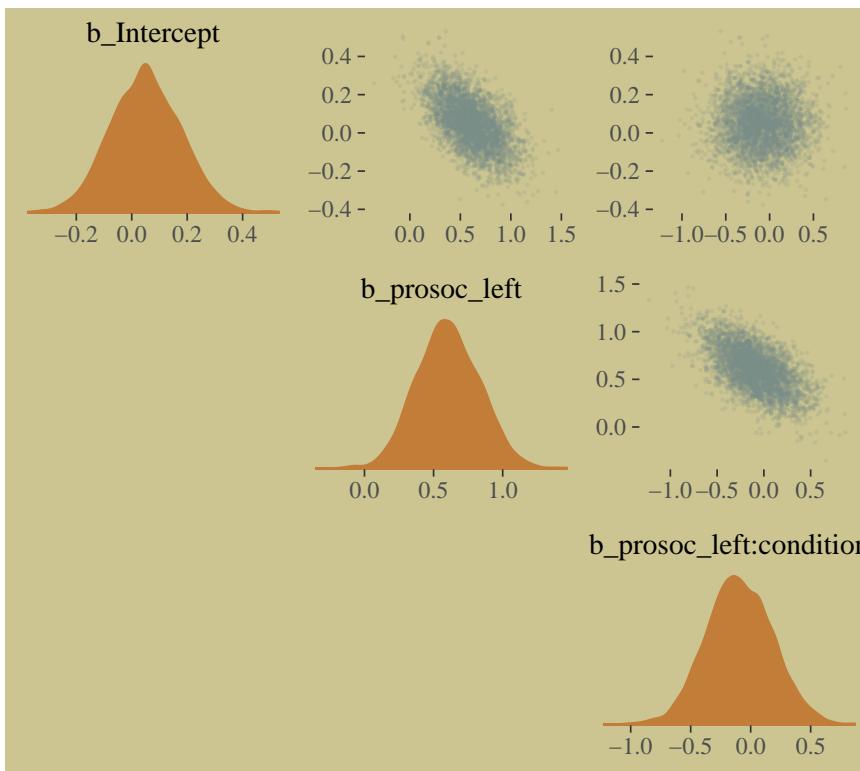
# the plot
ppa %>%
  ggplot(aes(x = x_axis)) +
  geom_smooth(aes(y = pulled_left, ymin = Q2.5, ymax = Q97.5, group = 0),
              stat = "identity",
              fill = wes_palette("Moonrise2")[2], color = "black",
              alpha = 1, size = 1/2) +
  geom_line(data = d_plot,
            aes(y = pulled_left, group = actor),
            color = wes_palette("Moonrise2")[1], size = 1/3) +
  scale_x_discrete(expand = c(.03, .03)) +
  coord_cartesian(ylim = 0:1) +
  labs(x = "prosoc_left/condition",
       y = "proportion pulled left") +
  theme(axis.ticks.x = element_blank())
```



McElreath didn't show the actual pairs plot in the text. Here's ours using `mcmc_pairs()`.

```
# this helps us set our custom color scheme
color_scheme_set(c(wes_palette("Moonrise2")[3],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[1]))

# the actual plot
mcmc_pairs(x = posterior_samples(b10.3),
            pars = c("b_Intercept", "b_prosoc_left", "b_prosoc_left:condition"),
            off_diag_args = list(size = 1/10, alpha = 1/6),
            diag_fun = "dens")
```



As McElreath observed, the posterior looks multivariate Gaussian.

In equations, the next model follows the form

$$\begin{aligned}
 \text{pulled_left}_i &\sim \text{Binomial}(1, p_i) \\
 \text{logit}(p_i) &= \alpha_{\text{actor}} + (\beta_1 + \beta_2 \text{condition}_i) \text{prosoc_left}_i \\
 \alpha_{\text{actor}} &\sim \text{Normal}(0, 10) \\
 \beta_1 &\sim \text{Normal}(0, 10) \\
 \beta_2 &\sim \text{Normal}(0, 10)
 \end{aligned}$$

Enclosing the `actor` variable within `factor()` will produce the indexing we need to get `actor`-specific intercepts. Also notice we're using the `0 + factor(actor)` part of the model `formula` to suppress the brms default intercept. As such, the priors for all parameters in the model will be of `class = b`. And since we're using the same Gaussian prior for each, we only need one line for the `prior` argument.

```
b10.4 <-  
  brm(data = d, family = binomial,  
        pulled_left | trials(1) ~ 0 + factor(actor) + prosoc_left + condition:prosoc_left ,  
        prior(normal(0, 10), class = b),  
        iter = 2500, warmup = 500, chains = 2, cores = 2,  
        control = list(adapt_delta = 0.9),  
        seed = 10)
```

Within the tidyverse, `distinct()` yields the information you'd otherwise get from `unique()`.

```
d %>%  
  distinct(actor)
```

```
##   actor  
## 1     1  
## 2     2  
## 3     3  
## 4     4  
## 5     5  
## 6     6  
## 7     7
```

We have no need to use something like `depth=2` for our posterior summary.

```
print(b10.4)
```

```
## Family: binomial  
## Links: mu = logit  
## Formula: pulled_left | trials(1) ~ 0 + factor(actor) + prosoc_left + condition:prosoc_left  
## Data: d (Number of observations: 504)  
## Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;  
##          total post-warmup samples = 4000  
##  
## Population-Level Effects:  
##                               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat  
## factoractor1            -0.74    0.27   -1.28   -0.21      3461 1.00  
## factoractor2             10.98    5.48    3.89   24.16      1870 1.00  
## factoractor3            -1.05    0.29   -1.63   -0.51      3714 1.00  
## factoractor4            -1.05    0.28   -1.61   -0.51      3870 1.00  
## factoractor5            -0.74    0.27   -1.28   -0.21      3782 1.00  
## factoractor6             0.22    0.27   -0.29    0.77      4221 1.00  
## factoractor7             1.82    0.40    1.08    2.69      4221 1.00  
## prosoc_left              0.84    0.26    0.35    1.35      2455 1.00  
## prosoc_left:condition   -0.13    0.30   -0.72    0.44      3259 1.00
```

```
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Correspondingly, `brms::posterior_samples()` returns an object for `b10.4` that doesn't quite follow the same structure as from `rethinking::extract.samples()`. We just have a typical 2-dimensional data frame.

```
post <- posterior_samples(b10.4)  
  
post %>%  
  glimpse()
```

```
## Observations: 4,000  
## Variables: 10  
## $ b_factoractor1      <dbl> -1.1858595, -0.4615217, -0.7064094, -0.2933396, -0.9631797, -1.0367221,...  
## $ b_factoractor2      <dbl> 7.454134, 6.917168, 7.084945, 7.208442, 16.542496, 5.172823, 8.026027, ...  
## $ b_factoractor3      <dbl> -0.7563732, -1.3653995, -0.4588494, -0.1941674, -1.6394630, -1.2838897,...  
## $ b_factoractor4      <dbl> -0.9358261, -0.8761278, -1.0917094, -1.0846353, -1.0000311, -1.4545123,...  
## $ b_factoractor5      <dbl> -0.2041299, -0.9987514, -0.3351474, -0.3790166, -0.9260228, -1.1574304,...  
## $ b_factoractor6      <dbl> 0.68325055, -0.24779408, 0.96302540, 0.98686825, -0.36103079, 0.6936556...  
## $ b_factoractor7      <dbl> 2.9389421, 2.5470499, 1.6366450, 1.2998254, 2.6914495, 1.3223443, 0.895...  
## $ b_prosoc_left       <dbl> 0.3384714, 0.8072413, 0.7844121, 1.0413892, 0.9789063, 1.3959586, 1.109...  
## $ `b_prosoc_left:condition` <dbl> 0.521824578, 0.147820067, -0.601883627, -0.619894359, 0.394578954, -0.3...  
## $ lp_...                <dbl> -297.4577, -292.0001, -293.5426, -300.4224, -296.1999, -292.4967, -294...
```

Our variant of Figure 10.3:

```
post %>%  
  ggplot(aes(x = b_factoractor2)) +  
    geom_density(color = "transparent",  
                 fill = wes_palette("Moonrise2")[1]) +  
    scale_y_continuous(NULL, breaks = NULL) +  
    labs(x      = NULL,  
         title   = "Actor 2's large and uncertain intercept",  
         subtitle = "Once your log-odds are above, like, 4, it's all\npretty much a probability of 1.")
```

Actor 2's large and uncertain intercept
Once your log-odds are above, like, 4, it's all pretty much a probability of 1.

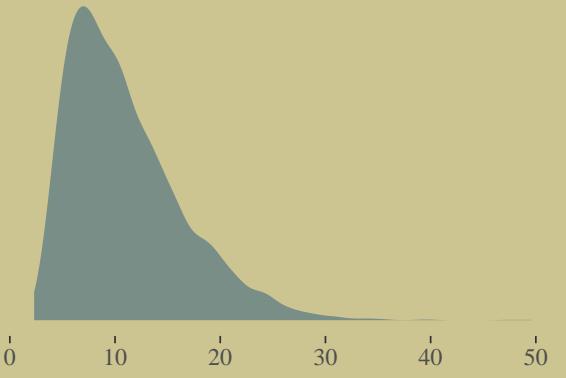
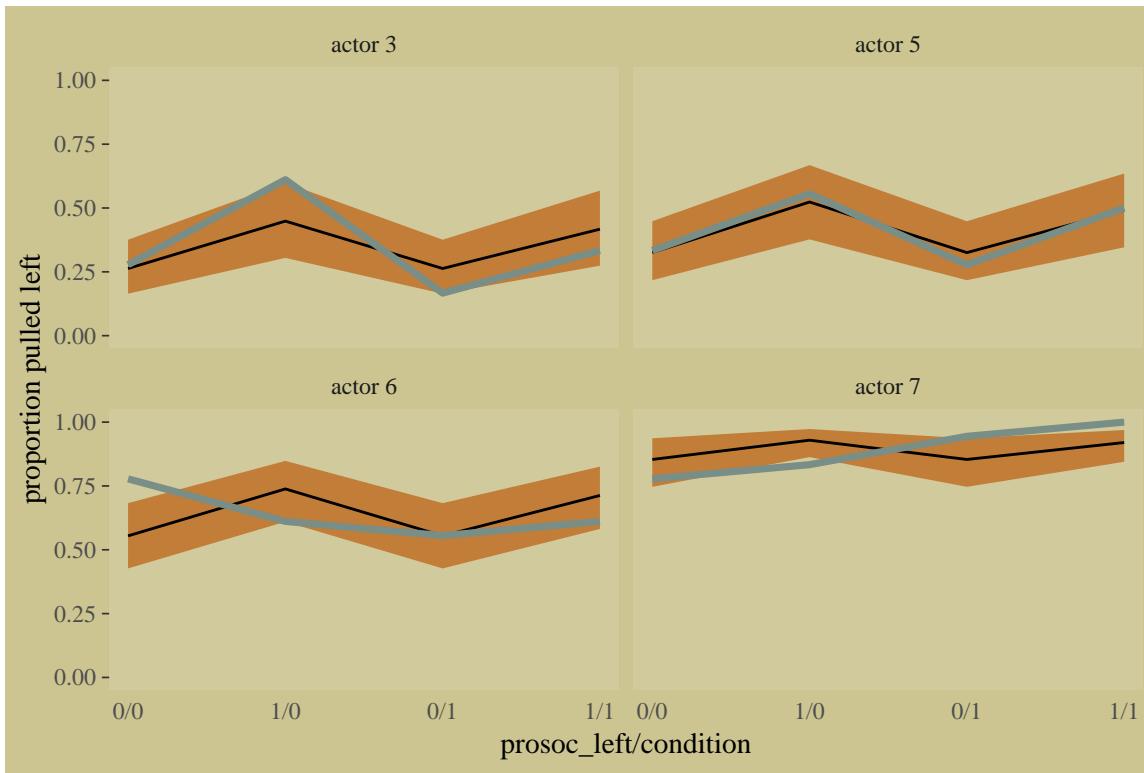


Figure 10.4. shows the idiographic trajectories for four of our chimps.

```
# subset the `d_plot` data
d_plot_4 <-
  d_plot %>%
  filter(actor %in% c(3, 5:7)) %>%
  ungroup() %>%
  mutate(actor = str_c("actor ", actor))

# compute the model-implied estimates with `fitted()` and wrangle
f <-
  fitted(b10.4) %>%
  as_tibble() %>%
  bind_cols(b10.4$data) %>%
  filter(actor %in% c(3, 5:7)) %>%
  distinct(Estimate, Q2.5, Q97.5, condition, prosoc_left, actor) %>%
  select(actor, everything()) %>%
  mutate(actor = str_c("actor ", actor),
        x_axis = str_c(prosoc_left, condition, sep = "/")) %>%
  mutate(x_axis = factor(x_axis, levels = c("0/0", "1/0", "0/1", "1/1"))) %>%
  rename(pulled_left = Estimate)

# plot
f %>%
  ggplot(aes(x = x_axis, y = pulled_left, group = actor)) +
  geom_smooth(aes(ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = wes_palette("Moonrise2")[2], color = "black",
              alpha = 1, size = 1/2) +
  geom_line(data = d_plot_4,
            color = wes_palette("Moonrise2")[1], size = 1.25) +
  scale_x_discrete(expand = c(.03, .03)) +
  coord_cartesian(ylim = 0:1) +
  labs(x = "prosoc_left/condition",
       y = "proportion pulled left") +
  theme(axis.ticks.x      = element_blank(),
        # color came from: http://www.color-hex.com/color/ccc591
        panel.background = element_rect(fill = "#d1ca9c",
                                         color = "transparent")) +
  facet_wrap(~actor)
```



10.1.1.1 Overthinking: Using the `by` `group_by()` function.

Let's work within the tidyverse, instead. If you wanted to compute the proportion of trials `pulled_left == 1` for each combination of `prosoc_left`, `condition`, and chimp `actor`, you'd put those last three variables within `group_by()` and then compute the `mean()` of `pulled_left` within `summarise()`.

```
d %>%
  group_by(prosoc_left, condition, actor) %>%
  summarise(`proportion pulled_left` = mean(pulled_left))
```

```
## # A tibble: 28 x 4
## # Groups: prosoc_left, condition [4]
##   prosoc_left condition actor `proportion pulled_left`
##       <int>      <int> <int>                  <dbl>
## 1         0          0     1                 0.333
## 2         0          0     2                 1.00 
## 3         0          0     3                 0.278
## 4         0          0     4                 0.333
## 5         0          0     5                 0.333
## 6         0          0     6                 0.778
## 7         0          0     7                 0.778
## 8         0          1     1                 0.278
## 9         0          1     2                 1.00 
## 10        0          1     3                 0.167
## # ... with 18 more rows
```

And since we're working within the tidyverse, that operation returns a tibble rather than a list.

10.1.2 Aggregated binomial: Chimpanzees again, condensed.

With the tidyverse, we use `group_by()` and `summarise()` to achieve what McElreath did with `aggregate()`.

```
d_aggregated <-
  d %>%
  select(-recipient, -block, -trial, -chose_prosoc) %>%
  group_by(actor, condition, prosoc_left) %>%
  summarise(x = sum(pulled_left))

d_aggregated %>%
  filter(actor %in% c(1, 2))
```

```
## # A tibble: 8 x 4
## # Groups:   actor, condition [4]
##   actor condition prosoc_left     x
##   <int>     <int>     <int> <int>
## 1     1         0         0     6
## 2     1         0         1     9
## 3     1         1         0     5
## 4     1         1         1    10
## 5     2         0         0    18
## 6     2         0         1    18
## 7     2         1         0    18
## 8     2         1         1    18
```

To fit an aggregated binomial model in brms, we augment the `<criterion> | trials()` syntax where the value that goes in `trials()` is either a fixed number, as in this case, or variable in the data indexing n . Either way, at least some of those trials will have an $n > 1$.

```
b10.5 <-
  brm(data = d_aggregated, family = binomial,
       x | trials(18) ~ 1 + prosoc_left + condition:prosoc_left,
       prior = c(prior(normal(0, 10), class = Intercept),
                  prior(normal(0, 10), class = b)),
       iter = 2500, warmup = 500, cores = 2, chains = 2,
       seed = 10)
```

We might compare `b10.3` with `b10.5` like this.

```
fixef(b10.3) %>% round(digits = 2)

##                               Estimate Est.Error Q2.5 Q97.5
## Intercept                   0.05     0.13 -0.19  0.30
## prosoc_left                  0.60     0.23  0.18  1.05
## prosoc_left:condition      -0.09     0.27 -0.62  0.45
```

```
fixef(b10.5) %>% round(digits = 2)

##                               Estimate Est.Error Q2.5 Q97.5
## Intercept                   0.04     0.13 -0.21  0.29
## prosoc_left                  0.61     0.23  0.18  1.06
## prosoc_left:condition      -0.09     0.26 -0.61  0.42
```

A coefficient plot can offer a complimentary perspective.

```
library(broom)

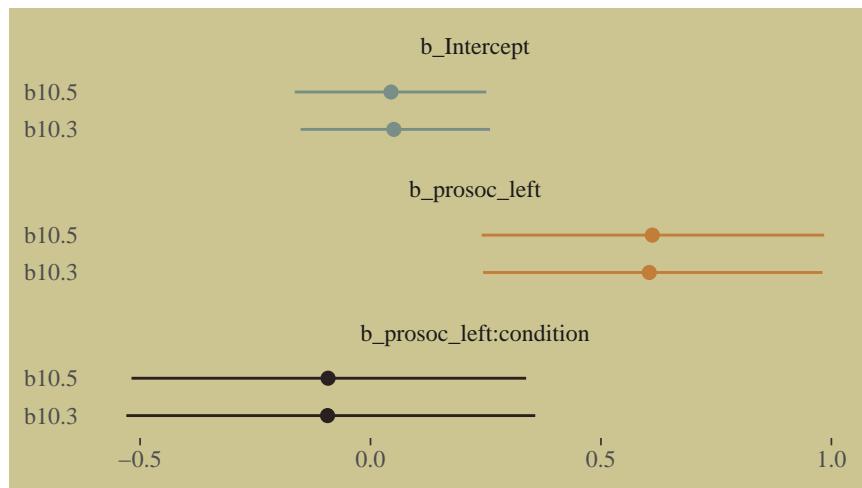
# wrangle
tibble(model  = str_c("b10.", c(3, 5))) %>%
  mutate(fit  = map(model, get)) %>%
```

```

mutate(tidy = map(fit, tidy)) %>%
unnest(tidy) %>%
filter(term != "lp__") %>%

# plot
ggplot() +
geom_pointrange(aes(x = model, y = estimate,
                     ymin = lower,
                     ymax = upper,
                     color = term),
                 shape = 16) +
scale_color_manual(values = wes_palette("Moonrise2")[c(1:2, 4)]) +
coord_flip() +
labs(x = NULL, y = NULL) +
theme(axis.ticks.y = element_blank(),
      legend.position = "none") +
facet_wrap(~term, ncol = 1)

```



The two are close within simulation error.

10.1.3 Aggregated binomial: Graduate school admissions.

Load the infamous UCBadmit data.

```

# detach(package:brms)
library(rethinking)
data(UCBadmit)
d <- UCBadmit

```

Switch from rethinking to brms.

```

detach(package:rethinking, unload = T)
library(brms)
rm(UCBadmit)

d

##   dept applicant.gender admit reject applications
## 1     A           male    512    313      825
## 2     A         female     89     19      108
## 3     B           male    353    207      560
## 4     B         female     17      8       25

```

```
## 5   C       male  120  205    325
## 6   C     female  202  391    593
## 7   D       male  138  279    417
## 8   D     female  131  244    375
## 9   E       male   53  138    191
## 10  E     female   94  299    393
## 11  F       male   22  351    373
## 12  F     female   24  317    341
```

Now compute our newly-constructed dummy variable, `male`.

```
d <-
d %>%
  mutate(male = ifelse(applicant.gender == "male", 1, 0))
```

The univariable logistic model with `male` as the sole predictor of `admit` follows the form

$$\begin{aligned} n_{\text{admit}_i} &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha + \beta \text{male}_i \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 10) \end{aligned}$$

The second model omits the `male` predictor.

```
b10.6 <-
  brm(data = d, family = binomial,
       admit | trials(applications) ~ 1 + male ,
       prior = c(prior(normal(0, 10), class = Intercept),
                  prior(normal(0, 10), class = b)),
       iter = 2500, warmup = 500, cores = 2, chains = 2,
       seed = 10)

b10.7 <-
  brm(data = d, family = binomial,
       admit | trials(applications) ~ 1,
       prior(normal(0, 10), class = Intercept),
       iter = 2500, warmup = 500, cores = 2, chains = 2,
       seed = 10)
```

Compute the information criteria for each model and save the results within the `brmfit` objects.

```
b10.6 <- add_criterion(b10.6, "waic")
b10.7 <- add_criterion(b10.7, "waic")
```

Here's the WAIC comparison.

```
w <- loo_compare(b10.6, b10.7, criterion = "waic")

print(w, simplify = F)

##      elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic    se_waic
## b10.6     0.0     0.0   -496.2      163.8    112.9     40.4   992.5    327.6
## b10.7   -29.7    82.1   -525.9      164.8     86.3     37.6  1051.8    329.7
```

If you prefer the difference in the WAIC metric, use our `cbind()` conversion method from above.

Bonus: Information criteria digression.

Let's see what happens if we switch to the LOO.

```
b10.6 <- add_criterion(b10.6, "loo")
```

```
## Warning: Found 7 observations with a pareto_k > 0.7 in model 'b10.6'. It is recommended to set 'reloo = TRUE'
## in order to calculate the ELPD without the assumption that these observations are negligible. This will ref
## the model 7 times to compute the ELPDs for the problematic observations directly.
```

```
b10.7 <- add_criterion(b10.7, "loo")
```

```
## Warning: Found 4 observations with a pareto_k > 0.7 in model 'b10.7'. It is recommended to set 'reloo = TRUE'
## in order to calculate the ELPD without the assumption that these observations are negligible. This will ref
## the model 4 times to compute the ELPDs for the problematic observations directly.
```

If you just ape the text and use the WAIC, everything appears fine. But holy smokes look at those nasty warning messages from the `loo()`! One of the frightening but ultimately handy things about working with the PSIS-LOO is that it requires we estimate a Pareto k parameter, which you can learn all about in the `loo`-package section of the [loo reference manual](#). As it turns out, the Pareto k [can be used as a diagnostic tool](#). Each case in the data gets its own k value and we like it when those ks are low. The makers of the `loo` package get worried when those ks exceed 0.7 and as a result, `loo()` spits out a warning message when they do.

First things first, if you explicitly open the `loo` package, you'll have access to some handy diagnostic functions.

```
library(loo)
```

We'll be leveraging those k values with the `pareto_k_table()` and `pareto_k_ids()` functions. Both functions take objects created by the `loo()` or `psis()` functions. So, before we can get busy, we'll first make two objects with the `loo()`.

```
l_b10.6 <- loo(b10.6)
```

```
## Warning: Found 7 observations with a pareto_k > 0.7 in model 'b10.6'. It is recommended to set 'reloo = TRUE'
## in order to calculate the ELPD without the assumption that these observations are negligible. This will ref
## the model 7 times to compute the ELPDs for the problematic observations directly.
```

```
l_b10.7 <- loo(b10.7)
```

```
## Warning: Found 4 observations with a pareto_k > 0.7 in model 'b10.7'. It is recommended to set 'reloo = TRUE'
## in order to calculate the ELPD without the assumption that these observations are negligible. This will ref
## the model 4 times to compute the ELPDs for the problematic observations directly.
```

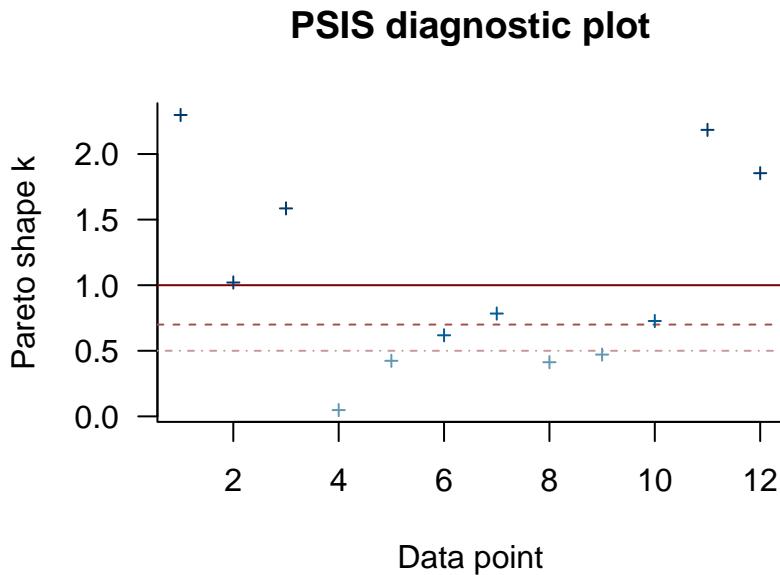
There are those warning messages, again. Using the `loo`-object for model `b10.6`, which we've named `l_b10.6`, let's take a look at the `pareto_k_table()` function.

```
pareto_k_table(l_b10.6)
```

```
## Pareto k diagnostic values:
#>          Count Pct.   Min. n_eff
#> (-Inf, 0.5] (good)    4   33.3%   635
#> (0.5, 0.7]  (ok)      1    8.3%   391
#> (0.7, 1]    (bad)     2   16.7%    90
#> (1, Inf)   (very bad) 5   41.7%     2
```

You may have noticed that this same table pops out when you just do something like `loo(b10.6)`. Recall that this data set has 12 observations (i.e., execute `count(d)`). With `pareto_k_table()`, we see how the Pareto k values have been categorized into bins ranging from “good” to “very bad”. Clearly, we like nice and low ks . In this example, our observations are all over the place, with 5 in the “bad” k range. We can take a closer look like this:

```
plot(l_b10.6)
```



So when you `plot()` a `loo` object, you get a nice diagnostic plot for those k values, ordered by observation number. Our plot indicates cases 1, 2, 3, 11, and 12 had “very bad” k values for this model. If we wanted to further verify to ourselves which observations those were, we’d use the `pareto_k_ids()` function.

```
pareto_k_ids(l_b10.6, threshold = 1)
```

```
## [1] 1 2 3 11 12
```

Note our use of the `threshold` argument. Play around with it to see how it works.

If you want an explicit look at those k values, you do:

```
l_b10.6$diagnostics
```

```
## $pareto_k
## [1] 2.29760490 1.02057506 1.58539693 0.04789929 0.42364825 0.61818010 0.78376601 0.41292102 0.47109654
## [10] 0.72698486 2.18412716 1.85392099
##
## $n_eff
## [1] 1.875425 45.003615 3.370704 3160.753814 1051.252892 391.491348 89.731920 1145.574819
## [9] 635.209403 258.696950 3.503268 7.730014
```

The `pareto_k` values can be used to examine cases that are overly-influential on the model parameters, something like a Cook’s D_i . See, for example [this discussion on stackoverflow.com](#) in which several members of the Stan team weighed in. The issue is also discussed in [this paper](#) and in [this presentation by Aki Vehtari](#).

Anyway, the implication of all this is these values suggest model `b10.6` isn’t a great fit for these data.

Part of the warning message for model `b10.6` read:

It is recommended to set ‘`reloo = TRUE`’ in order to calculate the ELPD without the assumption that these observations are negligible. This will refit the model $[n]$ times to compute the ELPDs for the problematic observations directly.

Let’s do that.

```
l_b10.6_reloo <- loo(b10.6, reloo = T)
```

Check the results.

```
l_b10.6_reloo

##
## Computed from 4000 by 12 log-likelihood matrix
##
##           Estimate    SE
## elpd_loo   -516.9 171.2
## p_loo      133.6  51.6
## looic     1033.8 342.3
## -----
## Monte Carlo SE of elpd_loo is NA.
##
## Pareto k diagnostic values:
##                               Count Pct. Min. n_eff
## (-Inf, 0.5]   (good)    11  91.7%  2
## (0.5, 0.7]   (ok)      1   8.3% 391
## (0.7, 1]     (bad)     0   0.0% <NA>
## (1, Inf)     (very bad) 0   0.0% <NA>
##
## All Pareto k estimates are ok (k < 0.7).
## See help('pareto-k-diagnostic') for details.
```

Now that looks better. We'll do the same thing for model b10.7.

```
l_b10.7_reloo <- loo(b10.7, reloo = T)
```

Okay, let's compare models with formal elpd_{loo} differences before and after adjusting with `reloo = T`.

```
loo_compare(l_b10.6, l_b10.7)
```

```
##       elpd_diff se_diff
## b10.6    0.0      0.0
## b10.7 -27.3     76.8
```

```
loo_compare(l_b10.6_reloo, l_b10.7_reloo)
```

```
##       elpd_diff se_diff
## b10.6    0.0      0.0
## b10.7 -17.5     82.3
```

In this case, the results are kinda similar. The standard errors for the differences are huge compared to the point estimates, suggesting large uncertainty. Watch out for this in your real-world data.

But this has all been a tangent from the central thrust of this section.

Back from our information criteria digression.

Let's get back on track with the text. Here's a look at `b10.6`, the unavailable model:

```
print(b10.6)
```

```

## Family: binomial
##   Links: mu = logit
## Formula: admit | trials(applications) ~ 1 + male
##   Data: d (Number of observations: 12)
## Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     -0.83      0.05   -0.93    -0.73        1929  1.00
## male          0.61      0.06    0.48     0.74        2527  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Here's the relative difference in admission odds.

```

fixef(b10.6)[2] %>%
  exp() %>%
  round(digits = 2)

```

```
## [1] 1.84
```

And now we'll compute difference in admission probabilities.

```

post <- posterior_samples(b10.6)

post %>%
  mutate(p_admit_male = inv_logit_scaled(b_Intercept + b_male),
         p_admit_female = inv_logit_scaled(b_Intercept),
         diff_admit = p_admit_male - p_admit_female) %>%
  summarise(`2.5%` = quantile(diff_admit, probs = .025),
            `50%` = median(diff_admit),
            `97.5%` = quantile(diff_admit, probs = .975))

##           2.5%       50%      97.5%
## 1 0.1125269 0.1414872 0.1705231

```

Instead of the `summarise()` code, we could have also used `tidybayes::median_qi(diff_admit)`. It's good to have options. Here's our version of Figure 10.5.

```

d <-
  d %>%
  mutate(case = factor(1:12))

p <-
  predict(b10.6) %>%
  as_tibble() %>%
  bind_cols(d)

d_text <-
  d %>%
  group_by(dept) %>%
  summarise(case = mean(as.numeric(case)),
            admit = mean(admit / applications) + .05)

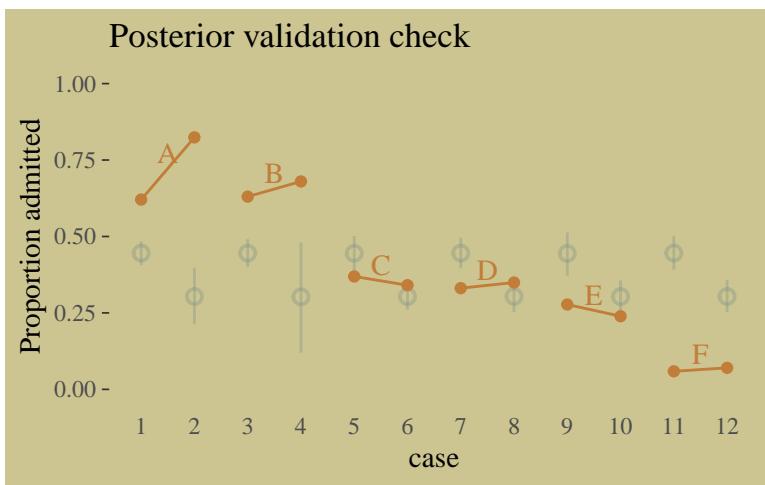
ggplot(data = d, aes(x = case, y = admit / applications)) +

```

```

geom_pointrange(data = p,
    aes(y      = Estimate / applications,
        ymin = Q2.5      / applications ,
        ymax = Q97.5     / applications),
    color = wes_palette("Moonrise2")[1],
    shape = 1, alpha = 1/3) +
geom_point(color = wes_palette("Moonrise2")[2]) +
geom_line(aes(group = dept),
    color = wes_palette("Moonrise2")[2]) +
geom_text(data = d_text,
    aes(y = admit, label = dept),
    color = wes_palette("Moonrise2")[2],
    family = "serif") +
coord_cartesian(ylim = 0:1) +
labs(y      = "Proportion admitted",
    title = "Posterior validation check") +
theme(axis.ticks.x = element_blank())

```



As alluded to in all that LOO/pareto_k talk, above, this is not a great fit. So we'll ditch the last model paradigm for one that answers the new question "*What is the average difference in probability of admission between females and males within departments?*" (p. 307). The statistical formula for the full model follows the form

$$\begin{aligned}
n_{\text{admit}_i} &\sim \text{Binomial}(n_i, p_i) \\
\text{logit}(p_i) &= \alpha_{\text{dept}_i} + \beta \text{male}_i \\
\alpha_{\text{dept}} &\sim \text{Normal}(0, 10) \\
\beta &\sim \text{Normal}(0, 10)
\end{aligned}$$

We don't need to coerce an index like McElreath did in the text. But here are the models.

```

b10.8 <-
  brm(data = d, family = binomial,
    admit | trials(applications) ~ 0 + dept,
    prior(normal(0, 10), class = b),
    iter = 2500, warmup = 500, cores = 2, chains = 2,
    seed = 10)

b10.9 <-
  update(b10.8,
    newdata = d,
    formula = admit | trials(applications) ~ 0 + dept + male)

```

Let's make two more `loo()` objects using `reloo = T`.

```
l_b10.8_reloo <- loo(b10.8, reloo = T)
l_b10.9_reloo <- loo(b10.9, reloo = T)
```

Now compare them.

```
loo_compare(l_b10.6_reloo, l_b10.7_reloo, l_b10.8_reloo, l_b10.9_reloo)
```

```
##      elpd_diff se_diff
## b10.8     0.0     0.0
## b10.9    -2.2     3.4
## b10.6 -453.8    167.2
## b10.7 -471.3    162.9
```

Here are the LOO weights.

```
model_weights(b10.6, b10.7, b10.8, b10.9,
               weights = "loo") %>%
  round(digits = 3)
```

```
## b10.6 b10.7 b10.8 b10.9
## 0.000 0.000 0.888 0.112
```

The parameters summaries for our multivariable model, `b10.9`, look like this:

```
fixef(b10.9) %>% round(digits = 2)

##           Estimate Est.Error Q2.5 Q97.5
## deptA      0.68     0.10  0.49   0.88
## deptB      0.65     0.12  0.41   0.88
## deptC     -0.58     0.07 -0.73  -0.44
## deptD     -0.61     0.09 -0.79  -0.44
## deptE     -1.06     0.10 -1.26  -0.86
## deptF     -2.64     0.16 -2.96  -2.32
## male       -0.10     0.08 -0.27   0.06
```

And on the proportional odds scale, the posterior mean for `b_male` is:

```
fixef(b10.9)[7, 1] %>% exp()
```

```
## [1] 0.9032744
```

Since we've been using brms, there's no need to fit our version of McElreath's `m10.9stan`. We already have that in our `b10.9`. But just for kicks and giggles, here's another way to get the model summary.

```
b10.9$fit
```

```
## Inference for Stan model: 90c98ca777a7f8ad7440192cd119b166.
## 2 chains, each with iter=2500; warmup=500; thin=1;
## post-warmup draws per chain=2000, total post-warmup draws=4000.
##
##           mean se_mean    sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## b_deptA  0.68    0.00 0.10  0.49   0.61   0.69   0.75   0.88  1864    1
## b_deptB  0.65    0.00 0.12  0.41   0.56   0.65   0.72   0.88  1754    1
## b_deptC -0.58    0.00 0.07 -0.73  -0.63  -0.58  -0.53  -0.44  3394    1
## b_deptD -0.61    0.00 0.09 -0.79  -0.67  -0.61  -0.55  -0.44  2795    1
```

```

## b_deptE -1.06  0.00 0.10 -1.26 -1.13 -1.06 -0.99 -0.86 3774 1
## b_deptF -2.64  0.00 0.16 -2.96 -2.74 -2.64 -2.53 -2.32 3314 1
## b_male   -0.10  0.00 0.08 -0.27 -0.16 -0.10 -0.05  0.06 1357 1
## lp__   -70.75  0.04 1.91 -75.43 -71.73 -70.41 -69.37 -68.04 1857 1
##
## Samples were drawn using NUTS(diag_e) at Sat Apr 20 12:59:27 2019.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).

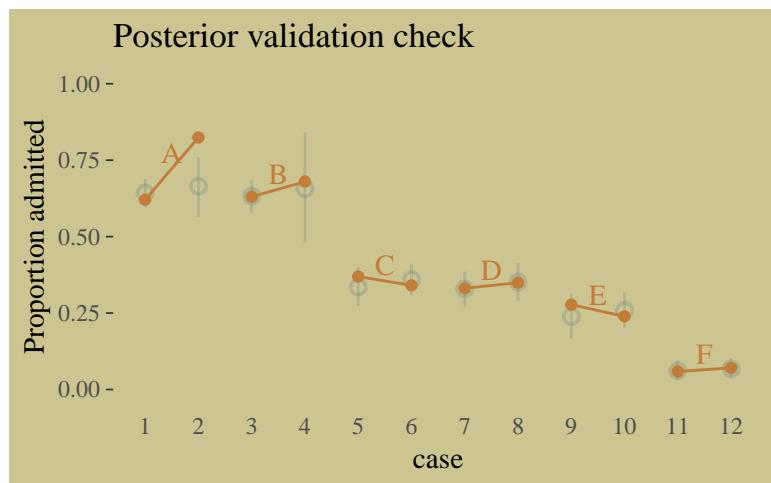
```

Here's our version of Figure 10.6, the posterior validation check.

```

predict(b10.9) %>%
  as_tibble() %>%
  bind_cols(d) %>%
  ggplot(aes(x = case, y = admit / applications)) +
  geom_pointrange(aes(y      = Estimate / applications,
                       ymin = Q2.5      / applications ,
                       ymax = Q97.5      / applications),
                  color = wes_palette("Moonrise2")[1],
                  shape = 1, alpha = 1/3) +
  geom_point(color = wes_palette("Moonrise2")[2]) +
  geom_line(aes(group = dept),
            color = wes_palette("Moonrise2")[2]) +
  geom_text(data = d_text,
            aes(y = admit, label = dept),
            color = wes_palette("Moonrise2")[2],
            family = "serif") +
  coord_cartesian(ylim = 0:1) +
  labs(y      = "Proportion admitted",
       title = "Posterior validation check") +
  theme(axis.ticks.x = element_blank())

```

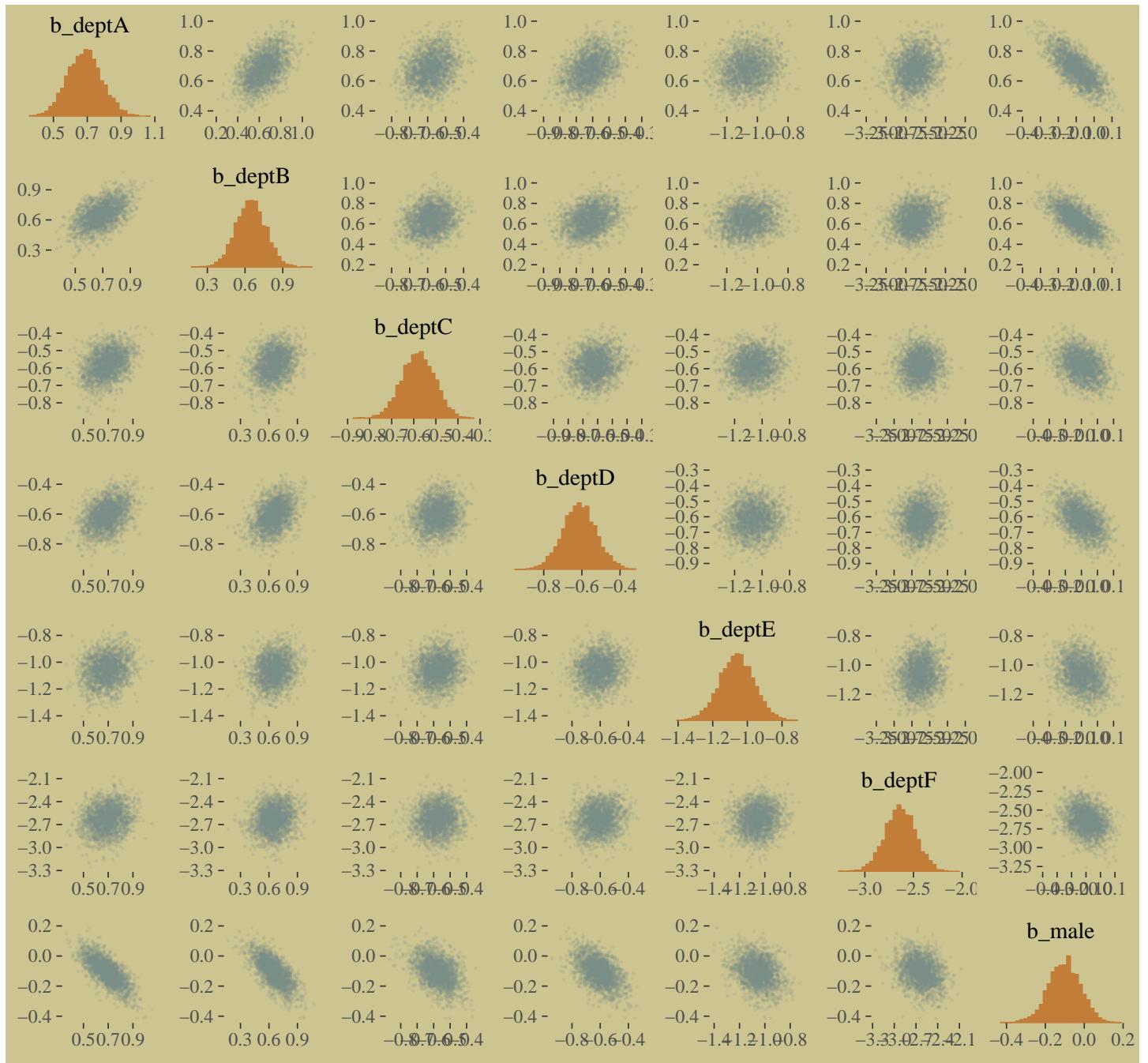


The model precisions are imperfect, but way more valid than before. The posterior looks reasonably multivariate Gaussian.

```

pairs(b10.9,
      off_diag_args = list(size = 1/10, alpha = 1/6))

```



10.1.3.1 Overthinking: WAIC and aggregated binomial models.

McElreath wrote:

The `WAIC` function in `rereThinking` detects aggregated binomial models and automatically splits them apart into 0/1 Bernoulli trials, for the purpose of calculating `WAIC`. It does this, because `WAIC` is computed point by point (see Chapter 6). So what you define as a “point” affects `WAIC`’s value. In an aggregated binomial each “point” is a bunch of independent trials that happen to share the same predictor values. In order for the disaggregated and aggregated models to agree, it makes sense to use the disaggregated representation. (p. 309)

To my knowledge, `brms::waic()` and `brms::loo()` do not do this, which might well be why some of our values didn’t match up with those in the text. If you have additional insight on this, please [share with the rest of the class](#).

10.1.4 Fitting binomial regressions with `glm()`.

We're not here to learn frequentist code, so we're going to skip most of this section. But model `b.good` is worth fitting. Here are the data.

```
# outcome and predictor almost perfectly associated
y <- c(rep(0, 10), rep(1, 10))
x <- c(rep(-1, 9), rep(1, 11))
```

Fit the `b.good` model.

```
b.good <-
  brm(data = list(y = y, x = x), family = binomial,
       y ~ 1 + x,
       prior = c(prior(normal(0, 10), class = Intercept),
                  prior(normal(0, 10), class = b)),
       seed = 10)
```

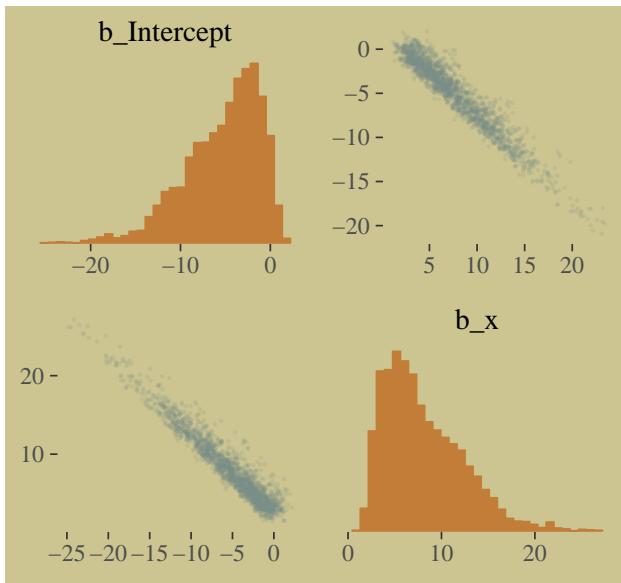
Our model summary will differ a bit from the one in the text. It seems this is because of the MAP/HMC contrast and our choice of priors.

```
print(b.good)
```

```
## Family: binomial
##   Links: mu = logit
## Formula: y ~ 1 + x
##   Data: list(y = y, x = x) (Number of observations: 20)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##                 Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     -5.34      4.24    -15.14     0.43        530 1.01
## x              8.07      4.23     2.47    18.06        532 1.01
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

You might experiment with different prior *SDs* to see how they influence the posterior *SDs*. Anyways, here's the `pairs()` plot McElreath excluded from the text.

```
pairs(b.good,
      off_diag_args = list(size = 1/10, alpha = 1/6))
```



That posterior, my friends, is not multivariate Gaussian. The plot deserves and extensive quote from McElreath.

Inspecting the pairs plot (not shown) demonstrates just how subtle even simple models can be, once we start working with GLMs. I don't say this to scare the reader. But it's true that even simple models can behave in complicated ways. How you fit the model is part of the model, and in principle no GLM is safe for MAP estimation. (p. 311)

10.2 Poisson regression

We'll simulate our sweet count data.

```
set.seed(10) # make the results reproducible

tibble(y = rbinom(1e5, 1000, 1/1000)) %>%
  summarise(y_mean      = mean(y),
            y_variance = var(y))

## # A tibble: 1 x 2
##   y_mean y_variance
##     <dbl>      <dbl>
## 1  0.994     0.995
```

Yes, those statistics are virtually the same. When dealing with Poisson data, $\mu = \sigma^2$. When you have a number of trials for which n is unknown or much larger than seen in the data, the Poisson likelihood is a useful tool. We define it like this

$$y \sim \text{Poisson}(\lambda)$$

As λ expresses both mean and variance because, within this model, the variance scales right along with the mean. Since λ is constrained to be positive, we typically use the log link. Thus the basic Poisson regression model is

$$\begin{aligned} y_i &\sim \text{Poisson}(\lambda_i) \\ \log(\lambda_i) &= \alpha + \beta x_i \end{aligned}$$

10.2.1 Example: Oceanic tool complexity.

Load the Kline data.

```
library(rethinking)
data(Kline)
d <- Kline
```

Switch from rethinking to brms.

```
detach(package:rethinking, unload = T)
library(brms)
rm(Kline)
```

```
d
```

	culture	population	contact	total_tools	mean_TU
## 1	Malekula	1100	low	13	3.2
## 2	Tikopia	1500	low	22	4.7
## 3	Santa Cruz	3600	low	24	4.0
## 4	Yap	4791	high	43	5.0
## 5	Lau Fiji	7400	high	33	5.0
## 6	Trobriand	8000	high	19	4.0
## 7	Chuuk	9200	high	40	3.8
## 8	Manus	13000	low	28	6.6
## 9	Tonga	17500	high	55	5.4
## 10	Hawaii	275000	low	71	6.6

Here are our new columns.

```
d <-
d %>%
  mutate(log_pop      = log(population),
        contact_high = ifelse(contact == "high", 1, 0))
```

Our statistical model will follow the form

$$\begin{aligned} \text{total_tools}_i &\sim \text{Poisson}(\lambda_i) \\ \log(\lambda_i) &= \alpha + \beta_1 \log_{\text{pop}}_i + \beta_2 \text{contact_high}_i + \beta_3 \text{contact_high}_i \times \log_{\text{pop}}_i \\ \alpha &\sim \text{Normal}(0, 100) \\ \beta_1 &\sim \text{Normal}(0, 1) \\ \beta_2 &\sim \text{Normal}(0, 1) \\ \beta_3 &\sim \text{Normal}(0, 1) \end{aligned}$$

The only new thing in our model code is `family = poisson`. brms defaults to the `log()` link.

```
b10.10 <-
  brm(data = d, family = poisson,
       total_tools ~ 1 + log_pop + contact_high + contact_high:log_pop,
       prior = c(prior(normal(0, 100), class = Intercept),
                 prior(normal(0, 1), class = b)),
       iter = 3000, warmup = 1000, chains = 4, cores = 4,
       seed = 10)

print(b10.10)
```

```
## Family: poisson
##   Links: mu = log
## Formula: total_tools ~ 1 + log_pop + contact_high + contact_high:log_pop
```

```

## Data: d (Number of observations: 10)
## Samples: 4 chains, each with iter = 3000; warmup = 1000; thin = 1;
##          total post-warmup samples = 8000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept      0.94     0.35    0.24    1.63      4087 1.00
## log_pop        0.26     0.03    0.20    0.33      4331 1.00
## contact_high   -0.11     0.84   -1.74    1.53      3115 1.00
## log_pop:contact_high  0.04     0.09   -0.14    0.22      3080 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Here's the lower triangle of the correlation matrix for the parameters.

```

post <- posterior_samples(b10.10)

post %>%
  select(-lp_) %>%
  rename(b_interaction = `b_log_pop:contact_high`) %>%
  psych::lowerCor()

##           b_Int b_lg_ b_cn_ b_ntr
## b_Intercept 1.00
## b_log_pop   -0.97  1.00
## b_contact_high -0.12  0.11  1.00
## b_interaction  0.06 -0.07 -0.99  1.00

```

And here's the coefficient plot via `bayesplot::mcmc_intervals()`:

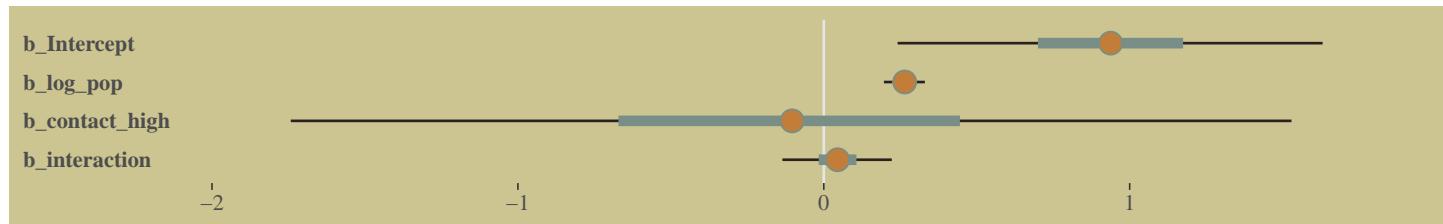
```

# we'll set a renewed color theme
color_scheme_set(c(wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[4],
                  wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[1]))

post %>%
  select(-lp_) %>%
  rename(b_interaction = `b_log_pop:contact_high`) %>%

  mcmc_intervals(prob = .5, prob_outer = .95) +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0))

```



How plausible is it a high-contact island will have more tools than a low-contact island?

```

post <-
  post %>%
    mutate(lambda_high = exp(b_Intercept + b_contact_high + (b_log_pop + `b_log_pop:contact_high`)*8),
          lambda_low = exp(b_Intercept + b_log_pop*8)) %>%
    mutate(diff = lambda_high - lambda_low)

post %>%
  summarise(sum = sum(diff > 0) / length(diff))

##      sum
## 1 0.95525

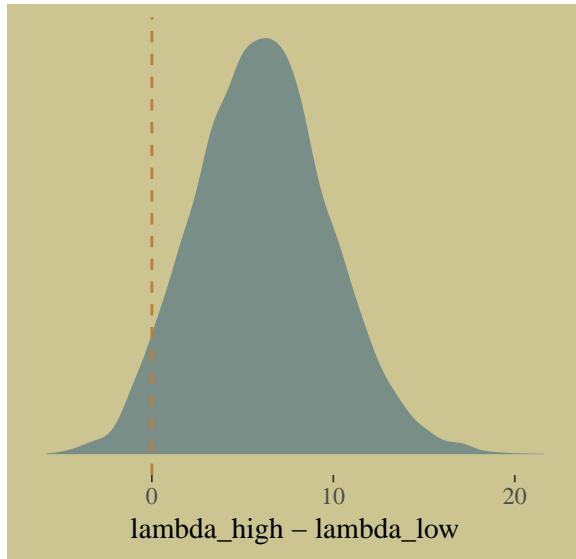
```

Quite, it turns out. Behold the corresponding Figure 10.8.a.

```

post %>%
  ggplot(aes(x = diff)) +
  geom_density(color = "transparent",
               fill = wes_palette("Moonrise2")[1]) +
  geom_vline(xintercept = 0, linetype = 2,
             color = wes_palette("Moonrise2")[2]) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(x = "lambda_high - lambda_low")

```



I'm not happy with how clunky this solution is, but one way to get those marginal dot and line portoin of the plot for the axes is to make intermediary tibbles. Anyway, here's a version of Figure 10.8.b.

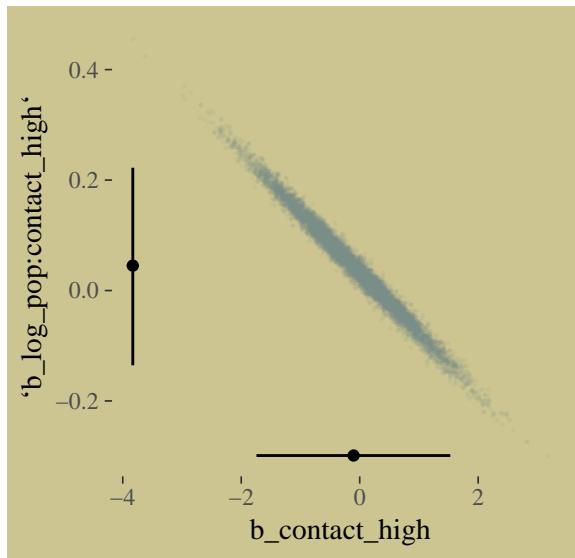
```

# intermediary tibbles for our the dot and line portoin of the plot
point_tibble <-
  tibble(x = c(median(post$b_contact_high), min(post$b_contact_high)),
         y = c(min(post$b_log_pop:contact_high), median(post$b_log_pop:contact_high)))

line_tibble <-
  tibble(parameter = rep(c("b_contact_high", "b_log_pop:contact_high"), each = 2),
         x = c(quantile(post$b_contact_high, probs = c(.025, .975)),
               rep(min(post$b_contact_high), times = 2)),
         y = c(rep(min(post$b_log_pop:contact_high), times = 2),
               quantile(post$b_log_pop:contact_high, probs = c(.025, .975))))

```

```
# the plot
post %>%
  ggplot(aes(x = b_contact_high, y = `b_log_pop:contact_high`)) +
  geom_point(color = wes_palette("Moonrise2")[1],
             size = 1/10, alpha = 1/10) +
  geom_point(data = point_tibble,
             aes(x = x, y = y)) +
  geom_line(data = line_tibble,
             aes(x = x, y = y, group = parameter))
```



Here we deconstruct model `b10.10`, bit by bit.

```
# no interaction
b10.11 <-
  update(b10.10, formula = total_tools ~ 1 + log_pop + contact_high)

# no contact rate
b10.12 <-
  update(b10.10, formula = total_tools ~ 1 + log_pop)

# no log-population
b10.13 <-
  update(b10.10, formula = total_tools ~ 1 + contact_high)

# intercept only
b10.14 <-
  update(b10.10, formula = total_tools ~ 1,
        seed = 10)
```

I know we got all excited with the LOO, above. Let's just be lazy and go WAIC. [Though beware, the LOO opens up a similar can of worms, here.]

```
b10.10 <- add_criterion(b10.10, criterion = "waic")
b10.11 <- add_criterion(b10.11, criterion = "waic")
b10.12 <- add_criterion(b10.12, criterion = "waic")
b10.13 <- add_criterion(b10.13, criterion = "waic")
b10.14 <- add_criterion(b10.14, criterion = "waic")
```

Now compare them.

```
w <- loo_compare(b10.10, b10.11, b10.12, b10.13, b10.14, criterion = "waic")

cbind(waic_diff = w[, 1] * -2,
      se       = w[, 2] * 2) %>%
      round(digits = 2)

##          waic_diff     se
## b10.11      0.00  0.00
## b10.10      0.75  1.35
## b10.12      4.86  8.29
## b10.14     62.07 34.55
## b10.13     71.30 47.08
```

Let's get those WAIC weights, too.

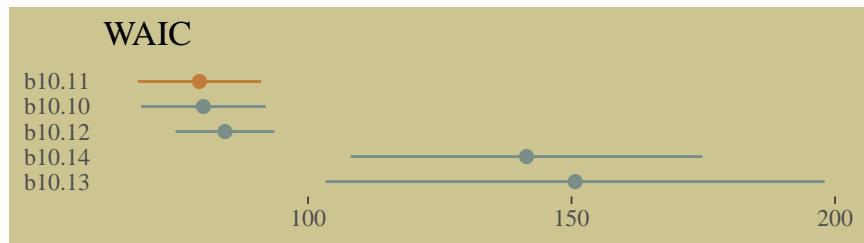
```
model_weights(b10.10, b10.11, b10.12, b10.13, b10.14, weights = "waic") %>%
  round(digits = 2)

## b10.10 b10.11 b10.12 b10.13 b10.14
##   0.39   0.56   0.05   0.00   0.00
```

Now wrangle `w` a little and make the WAIC plot.

```
w %>%
  data.frame() %>%
  rownames_to_column(var = "model") %>%

  ggplot(aes(x = reorder(model, -waic),
              y      = waic,
              ymin = waic - se_waic,
              ymax = waic + se_waic,
              color = model)) +
  geom_pointrange(shape = 16, show.legend = F) +
  scale_color_manual(values = wes_palette("Moonrise2")[c(1, 2, 1, 1, 1)]) +
  coord_flip() +
  labs(x = NULL, y = NULL,
       title = "WAIC") +
  theme(axis.ticks.y    = element_blank())
```



Here's our version of Figure 10.9. Recall, to do an “ensemble” posterior prediction in brms, one uses the `pp_average()` function. I know we were just lazy and focused on the WAIC. But let's play around, a bit. Here we'll weight the models based on the LOO by adding a `weights = "loo"` argument to the `pp_average()` function. If you check the corresponding section of the [brms reference manual](#), you'll find several weighting schemes.

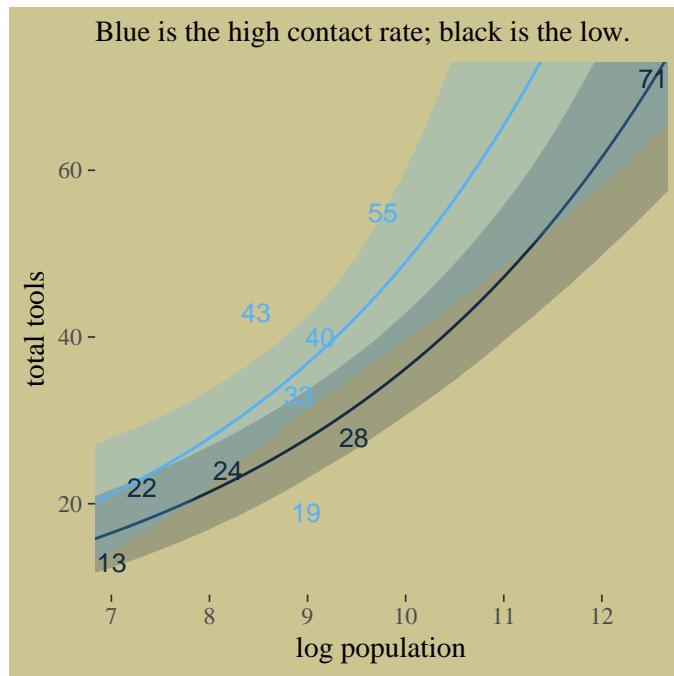
```
nd <-
  tibble(contact_high = 0:1) %>%
  expand(contact_high,
        log_pop = seq(from = 6.5, to = 13, length.out = 50))
```

```

ppa <-
  pp_average(b10.10, b10.11, b10.12,
              weights = "loo",
              method   = "fitted",
              newdata  = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

ppa %>%
  ggplot(aes(x      = log_pop,
             group = contact_high)) +
  geom_smooth(aes(y = Estimate, ymin = Q2.5, ymax = Q97.5,
                  fill = contact_high, color = contact_high),
               stat = "identity",
               alpha = 1/4, size = 1/2) +
  geom_text(data = d,
            aes(y      = total_tools,
                 label = total_tools,
                 color = contact_high),
            size = 3.5) +
  coord_cartesian(xlim = c(7.1, 12.4),
                  ylim = c(12, 70)) +
  labs(x = "log population",
       y = "total tools",
       subtitle = "Blue is the high contact rate; black is the low.") +
  theme(legend.position = "none",
        panel.border    = element_blank())

```



In case you were curious, here are those LOO weights:

```

model_weights(b10.10, b10.11, b10.12,
              weights = "loo")

```

```

##     b10.10     b10.11     b10.12
## 0.37857139 0.57188875 0.04953986

```

10.2.2 MCMC islands.

We fit our analogue to `m10.10stan`, `b10.10`, some time ago.

```
print(b10.10)

## Family: poisson
## Links: mu = log
## Formula: total_tools ~ 1 + log_pop + contact_high + contact_high:log_pop
## Data: d (Number of observations: 10)
## Samples: 4 chains, each with iter = 3000; warmup = 1000; thin = 1;
##          total post-warmup samples = 8000
##
## Population-Level Effects:
##                               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  0.94     0.35    0.24    1.63      4087 1.00
## log_pop                     0.26     0.03    0.20    0.33      4331 1.00
## contact_high                -0.11    0.84   -1.74    1.53      3115 1.00
## log_pop:contact_high        0.04     0.09   -0.14    0.22      3080 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Center `log_pop`.

```
d <-
  d %>%
  mutate(log_pop_c = log_pop - mean(log_pop))
```

Now fit the `log_pop`-centered model.

```
b10.10_c <-
  brm(data = d, family = poisson,
       total_tools ~ 1 + log_pop_c + contact_high + contact_high:log_pop_c,
       prior = c(prior(normal(0, 10), class = Intercept),
                 prior(normal(0, 1), class = b)),
       iter = 3000, warmup = 1000, chains = 4, cores = 4,
       seed = 10)
```

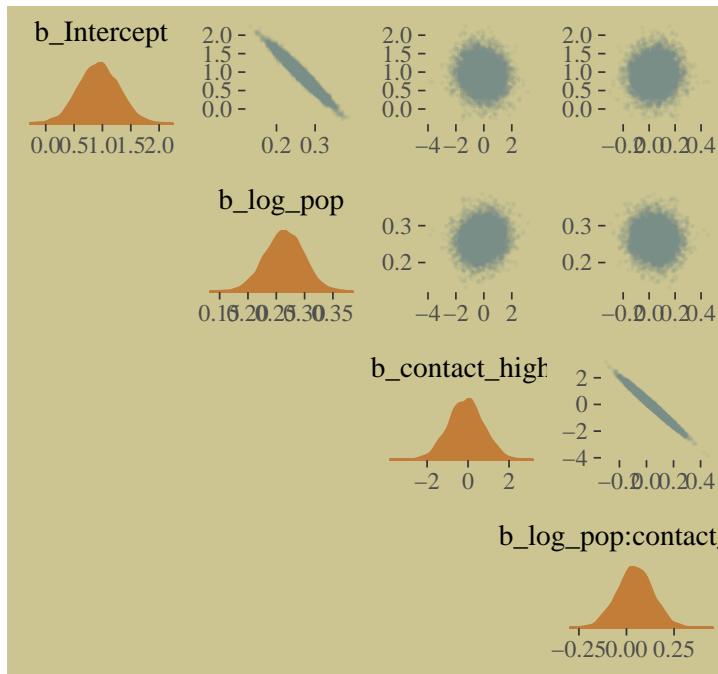
```
print(b10.10_c)
```

```
## Family: poisson
## Links: mu = log
## Formula: total_tools ~ 1 + log_pop_c + contact_high + contact_high:log_pop_c
## Data: d (Number of observations: 10)
## Samples: 4 chains, each with iter = 3000; warmup = 1000; thin = 1;
##          total post-warmup samples = 8000
##
## Population-Level Effects:
##                               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept                  3.31     0.09    3.13    3.49      6035 1.00
## log_pop_c                   0.26     0.04    0.19    0.33      6024 1.00
## contact_high                0.28     0.12    0.06    0.52      6606 1.00
## log_pop_c:contact_high     0.06     0.17   -0.26    0.38      6718 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

We'll use `mcmc_pairs()`, again, for Figure 10.10.a.

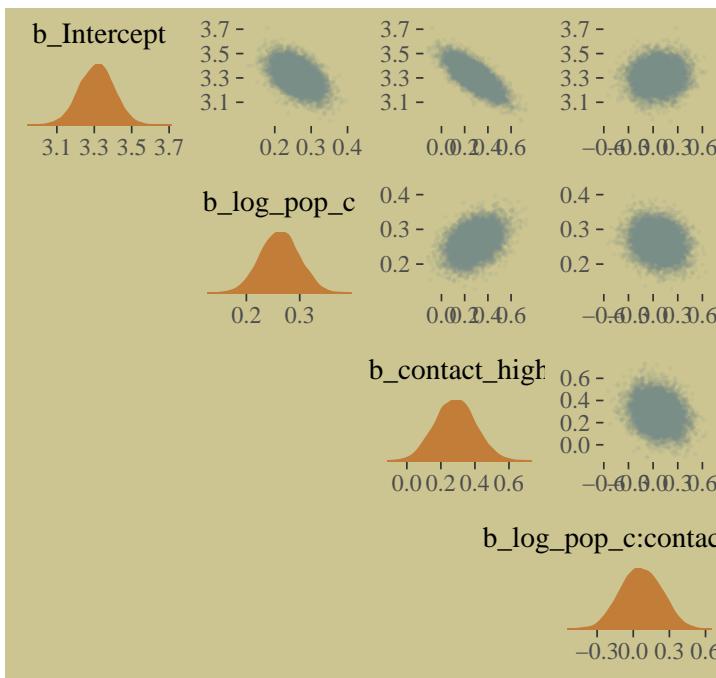
```
# this helps us set our custom color scheme
color_scheme_set(c(wes_palette("Moonrise2")[3],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[2],
                  wes_palette("Moonrise2")[1],
                  wes_palette("Moonrise2")[1]))

# the actual plot
mcmc_pairs(x = posterior_samples(b10.10),
            pars = c("b_Intercept", "b_log_pop", "b_contact_high", "b_log_pop:contact_high"),
            off_diag_args = list(size = 1/10, alpha = 1/10),
            diag_fun = "dens")
```



And now behold Figure 10.10.b.

```
mcmc_pairs(x = posterior_samples(b10.10_c),
            pars = c("b_Intercept", "b_log_pop_c", "b_contact_high", "b_log_pop_c:contact_high"),
            off_diag_args = list(size = 1/10, alpha = 1/10),
            diag_fun = "dens")
```



If you really want the correlation point estimates, use `psych::lowerCorr()`.

```
psych::lowerCor(posterior_samples(b10.10) [, 1:4])
```

```
##                                     b_Int b_lg_ b_cn_ b__:_  
## b_Intercept                  1.00  
## b_log_pop                 -0.97  1.00  
## b_contact_high              -0.12  0.11  1.00  
## b_log_pop:contact_high     0.06 -0.07 -0.99  1.00
```

```
psych::lowerCor(posterior_samples(b10.10_c) [, 1:4])
```

```
##                                     b_Int b_lg_ b_cn_ b__:_  
## b_Intercept                  1.00  
## b_log_pop_c                -0.48  1.00  
## b_contact_high              -0.77  0.36  1.00  
## b_log_pop_c:contact_high   0.09 -0.19 -0.24  1.00
```

10.2.3 Example: Exposure and the offset.

For the last Poisson example, we'll look at a case where the exposure varies across observations. When the length of observation, area of sampling, or intensity of sampling varies, the counts we observe also naturally vary. Since a Poisson distribution assumes that the rate of events is constant in time (or space), it's easy to handle this. All we need to do, as explained on page 312 [of the text], is to add the logarithm of the exposure to the linear model. The term we add is typically called an *offset*. (p. 321, *emphasis* in the original)

Here we simulate our data.

```
set.seed(10)  
  
num_days  <- 30  
y         <- rpois(num_days, 1.5)  
  
num_weeks <- 4  
y_new     <- rpois(num_weeks, 0.5 * 7)
```

Let's make them tidy and add `log_days`.

```
(  
  d <-  
  tibble(y      = c(y, y_new),  
         days    = c(rep(1, num_days), rep(7, num_weeks)),  
         monastery = c(rep(0, num_days), rep(1, num_weeks))) %>%  
  mutate(log_days = log(days))  
)  
  
## # A tibble: 34 x 4  
##       y   days  monastery log_days  
##   <int> <dbl>     <dbl>     <dbl>  
## 1     1     1         0        0  
## 2     1     1         0        0  
## 3     1     1         0        0  
## 4     2     1         0        0  
## 5     0     1         0        0  
## 6     1     1         0        0  
## 7     1     1         0        0  
## 8     1     1         0        0  
## 9     2     1         0        0  
## 10    1     1         0        0  
## # ... with 24 more rows
```

With the `brms` package, you use the `offset()` syntax, in which you put a pre-processed variable like `log_days` or the log of a variable, such as `log(days)`.

```
b10.15 <-  
  brm(data = d, family = poisson,  
        y ~ 1 + offset(log_days) + monastery,  
        prior = c(prior(normal(0, 100), class = Intercept),  
                  prior(normal(0, 1), class = b)),  
        iter = 2500, warmup = 500, cores = 2, chains = 2,  
        seed = 10)
```

The model summary:

```
print(b10.15)  
  
##  Family: poisson  
##  Links: mu = log  
##  Formula: y ~ 1 + offset(log_days) + monastery  
##  Data: d (Number of observations: 34)  
##  Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;  
##            total post-warmup samples = 4000  
##  
##  Population-Level Effects:  
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
##  Intercept      0.30      0.16     -0.04     0.61        3168  1.00  
##  monastery     -1.10      0.31     -1.72     -0.52        3340  1.00  
##  
##  Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
##  is a crude measure of effective sample size, and Rhat is the potential  
##  scale reduction factor on split chains (at convergence, Rhat = 1).
```

The model summary helps clarify that when you use `offset()`, `brm()` fixes the value. Thus there is no parameter estimate for the `offset()`. It's a fixed part of the model not unlike the ν parameter of the Student- t distribution gets fixed to infinity when you use the Gaussian likelihood.

Here we'll compute the posterior means and 89% HDIs with `tidybayes::mean_hdi()`.

```
library(tidybayes)

posterior_samples(b10.15) %>%
  transmute(lambda_old = exp(b_Intercept),
            lambda_new = exp(b_Intercept + b_monastery)) %>%
  gather() %>%
  mutate(key = factor(key, levels = c("lambda_old", "lambda_new"))) %>%
  group_by(key) %>%
  mean_hdi(value, .width = .89) %>%
  mutate_if(is.double, round, digits = 2)

## # A tibble: 2 x 7
##   key      value .lower .upper .width .point .interval
##   <fct>    <dbl>  <dbl>  <dbl>  <dbl>  <chr>   <chr>
## 1 lambda_old  1.37   0.99   1.7   0.89 mean     hdi
## 2 lambda_new  0.47   0.27   0.66   0.89 mean     hdi
```

As McElreath pointed out in the text, “Your estimates will be slightly different, because you got different randomly simulated data” (p. 322).

10.3 Other count regressions

The next two of the remaining four models are maximum entropy distributions for certain problem types. The last two are mixtures, of which we'll see more in the next chapter.

10.3.1 Multinomial.

When more than two types of unordered events are possible, and the probability of each type of event is constant across trials, then the maximum entropy distribution is the multinomial distribution. [We] already met the multinomial, implicitly, in Chapter 9 when we tossed pebbles into buckets as an introduction to maximum entropy. The binomial is really a special case of this distribution. And so its distribution formula resembles the binomial, just extrapolated out to three or more types of events. If there are K types of events with probabilities p_1, \dots, p_K , then the probability of observing y_1, \dots, y_K events of each type out of n trials is (p. 323):

$$\Pr(y_1, \dots, y_K | n, p_1, \dots, p_K) = \frac{n!}{\prod_i y_i!} \prod_{i=1}^K p_i^{y_i}$$

Compare that equation with the simpler version in section 2.3.1 (page 33 in the text).

10.3.1.1 Explicit multinomial models.

“The conventional and natural link is this context is the *multinomial logit*. This link function takes a vector of *scores*, one for each K event types, and computed the probability of a particular type of event K as” (p. 323, *emphasis* in the original)

$$\Pr(k | s_1, s_2, \dots, s_K) = \frac{\exp(s_k)}{\sum_{i=1}^K \exp(s_i)}$$

Let's simulate the data.

```

library(rethinking)

# simulate career choices among 500 individuals
n      <- 500          # number of individuals
income <- 1:3           # expected income of each career
score  <- 0.5 * income # scores for each career, based on income

# next line converts scores to probabilities
p <- softmax(score[1], score[2], score[3])

# now simulate choice
# outcome career holds event type values, not counts
career <- rep(NA, n)  # empty vector of choices for each individual

set.seed(10)
# sample chosen career for each individual
for(i in 1:n) career[i] <- sample(1:3, size = 1, prob = p)

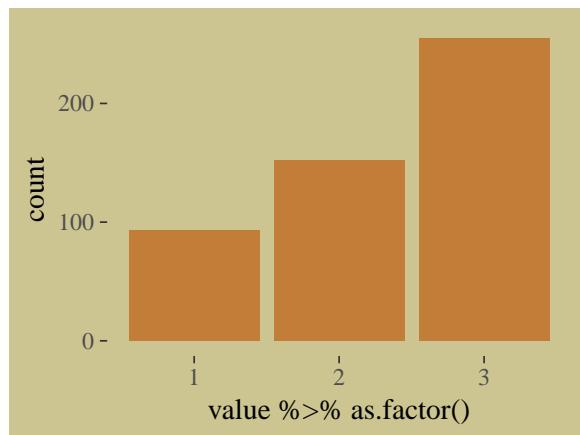
```

Here's what the data look like.

```

career %>%
  as_tibble() %>%
  ggplot(aes(x = value %>% as.factor())) +
  geom_bar(size = 0, fill = wes_palette("Moonrise2")[2])

```



Switch out rethinking for brms.

```

detach(package:rethinking, unload = T)
library(brms)

```

Here's my naïve attempt to fit the model in brms.

```

b10.16 <-
  brm(data = list(career = career),
       family = categorical(link = logit),
       career ~ 1,
       prior(normal(0, 5), class = Intercept),
       iter = 2500, warmup = 500, cores = 2, chains = 2,
       seed = 10)

```

This differs from McElreath's m10.16. Most obviously, this has two parameters. McElreath's m10.16 only has one. If you have experience with these models and know how to reproduce McElreath's results in brms, [please share your code](#).

```
print(b10.16)

## Family: categorical
## Links: mu2 = logit; mu3 = logit
## Formula: career ~ 1
## Data: list(career = career) (Number of observations: 500)
## Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## mu2_Intercept     0.49      0.14    0.23    0.76        1220 1.00
## mu3_Intercept     1.01      0.12    0.77    1.25        1263 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Here's the second data simulation, this time based on McElreath's R code 10.58.

```
library(rethinking)

n <- 100

set.seed(10)
# simulate family incomes for each individual
family_income <- runif(n)

# assign a unique coefficient for each type of event
b <- (1:-1)
career <- rep(NA, n) # empty vector of choices for each individual

for (i in 1:n) {
  score <- 0.5 * (1:3) + b * family_income[i]
  p <- softmax(score[1], score[2], score[3])
  career[i] <- sample(1:3, size = 1, prob = p)
}
```

Switch out rethinking for brms.

```
detach(package:rethinking, unload = T)
library(brms)
```

Here's the brms version of McElreath's m10.17.

```
b10.17 <-
  brm(data = list(career = career, # note how we used a list instead of a tibble
                  family_income = family_income),
       family = categorical(link = logit),
       career ~ 1 + family_income,
       prior = c(prior(normal(0, 5), class = Intercept),
                 prior(normal(0, 5), class = b)),
       iter = 2500, warmup = 500, cores = 2, chains = 2,
       seed = 10)
```

Happily, these results cohere with the rethinking model.

```
print(b10.17)

## Family: categorical
## Links: mu2 = logit; mu3 = logit
## Formula: career ~ 1 + family_income
## Data: list(career = career, family_income = family_incom (Number of observations: 100)
## Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;
##          total post-warmup samples = 4000
##
## Population-Level Effects:
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## mu2_Intercept            1.03     0.55    0.02    2.11      2758 1.00
## mu3_Intercept            1.04     0.54    0.02    2.13      2753 1.00
## mu2_family_income       -1.77     1.00   -3.72    0.16      2931 1.00
## mu3_family_income       -1.72     0.99   -3.76    0.13      2846 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

McElreath described the parameters as “on a scale that is very hard to interpret” (p. 325). Indeed.

10.3.1.2 Multinomial in disguise as Poisson.

Here we fit a multinomial likelihood by refactoring it to a series of Poissons. Let’s retrieve the Berkeley data.

```
library(rethinking)

data(UCBadmit)
d <- UCBadmit
rm(UCBadmit)

detach(package:rethinking, unload = T)
library(brms)
```

Fit the models.

```
# binomial model of overall admission probability
b_binom <-
  brm(data = d, family = binomial,
       admit | trials(applications) ~ 1,
       prior(normal(0, 100), class = Intercept),
       iter = 2000, warmup = 1000, cores = 3, chains = 3,
       seed = 10)

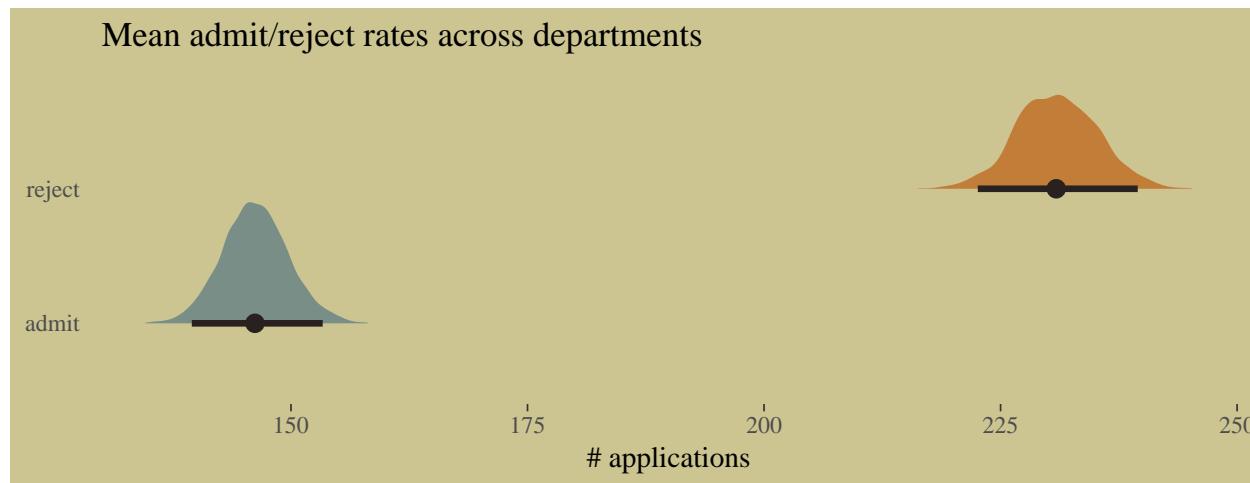
# Poisson model of overall admission rate and rejection rate
b_pois <-
  brm(data = d %>%
        mutate(rej = reject), # 'reject' is a reserved word
        family = poisson,
        mvbind(admit, rej) ~ 1,
        prior(normal(0, 100), class = Intercept),
        iter = 2000, warmup = 1000, cores = 3, chains = 3,
        seed = 10)
```

Note, the `mvbind()` syntax made `b_pois` a multivariate Poisson model. Starting with version 2.0.0, `brms` supports a variety of [multivariate models](#). Anyway, here are the implications of `b_pois`.

```
# extract the samples
post <- posterior_samples(b_pois)

# wrangle
post %>%
  transmute(admit = exp(b_admit_Intercept),
            reject = exp(b_rej_Intercept)) %>%
  gather() %>%

# plot
ggplot(aes(x = value, y = key, fill = key)) +
  geom_halfeyeh(point_interval = median_qi, .width = .95,
                color = wes_palette("Moonrise2")[4]) +
  scale_fill_manual(values = c(wes_palette("Moonrise2")[1],
                               wes_palette("Moonrise2")[2])) +
  labs(title = "Mean admit/reject rates across departments",
       x     = "# applications",
       y     = NULL) +
  theme(legend.position = "none",
        axis.ticks.y = element_blank())
```



The model summaries:

```
print(b_binom)

## # Family: binomial
## # Links: mu = logit
## # Formula: admit | trials(applications) ~ 1
## # Data: d (Number of observations: 12)
## # Samples: 3 chains, each with iter = 2000; warmup = 1000; thin = 1;
## #          total post-warmup samples = 3000
##
## # Population-Level Effects:
## #           Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## # Intercept    -0.46      0.03    -0.52     -0.40        1219  1.00
##
## # Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## # is a crude measure of effective sample size, and Rhat is the potential
## # scale reduction factor on split chains (at convergence, Rhat = 1).
```

```
print(b_pois)

## # Family: MV(poisson, poisson)
```

```

##   Links: mu = log
##           mu = log
## Formula: admit ~ 1
##           rej ~ 1
## Data: d %>% mutate(rej = reject) (Number of observations: 12)
## Samples: 3 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 3000
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## admit_Intercept     4.98      0.02    4.94    5.03        2194 1.00
## rej_Intercept       5.44      0.02    5.41    5.48        2591 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Here's the posterior mean for the probability of admission, based on `b_binom`.

```

fixef(b_binom)[ , "Estimate"] %>%
  inv_logit_scaled()

```

```
## [1] 0.387497
```

Happily, we get the same value within simulation error from model `b_pois`.

```

k <-
  fixef(b_pois) %>%
  as.numeric()

exp(k[1]) / (exp(k[1]) + exp(k[2]))

```

```
## [1] 0.3876675
```

The formula for what we just did in code is

$$p_{\text{admit}} = \frac{\lambda_1}{\lambda_1 + \lambda_2} = \frac{\exp(\alpha_1)}{\exp(\alpha_1) + \exp(\alpha_2)}$$

10.3.2 Geometric.

Sometimes a count variable is a number of events up until something happened. Call this “something” the terminating event. Often we want to model the probability of that event, a kind of analysis known as event history analysis or survival analysis. When the probability of the terminating event is constant through time (or distance), and the units of time (or distance) are discrete, a common likelihood function is the geometric distribution. This distribution has the form:

$$\Pr(y|p) = p(1-p)^{y-1}$$

where y is the number of time steps (events) until the terminating event occurred and p is the probability of that event in each time step. This distribution has maximum entropy for unbounded counts with constant expected value. (pp. 327–328)

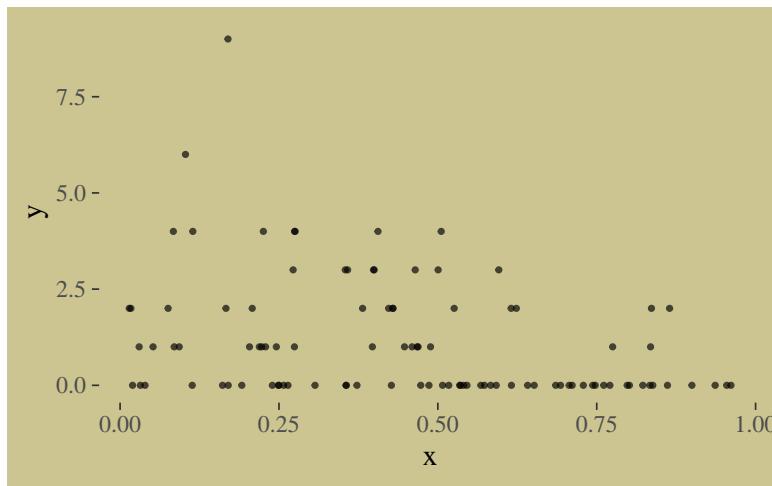
Here we simulate exemplar data.

```
# simulate
n <- 100
set.seed(10)
x <- runif(n)

set.seed(10)
y <- rgeom(n, prob = inv_logit_scaled(-1 + 2 * x))
```

In case you're curious, here are the data.

```
list(y = y, x = x) %>%
  as_tibble() %>%
  ggplot(aes(x = x, y = y)) +
  geom_point(size = 3/5, alpha = 2/3)
```



We fit the geometric model using `family = geometric(link = log)`.

```
b10.18 <-
  brm(data = list(y = y, x = x),
       family = geometric(link = log),
       y ~ 0 + intercept + x,
       prior = c(prior(normal(0, 10), class = b, coef = intercept),
                 prior(normal(0, 1), class = b)),
       iter = 2500, warmup = 500, chains = 2, cores = 2,
       seed = 10)
```

The results:

```
print(b10.18, digits = 2)

##  Family: geometric
##  Links: mu = log
##  Formula: y ~ 0 + intercept + x
##  Data: list(y = y, x = x) (Number of observations: 100)
##  Samples: 2 chains, each with iter = 2500; warmup = 500; thin = 1;
##            total post-warmup samples = 4000
##
##  Population-Level Effects:
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##  intercept      0.75      0.24     0.26     1.20        1322  1.00
##  x             -1.62      0.51    -2.58    -0.61        1286  1.00
##
```

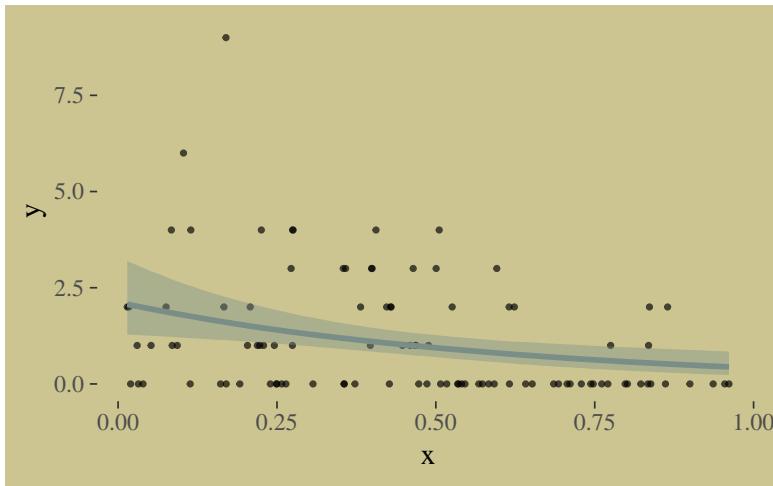
```
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

It turns out brms uses a [different parameterization for the geometric distribution](#) than rethinking does. It follows the form

$$f(y_i) = \binom{y_i}{y_i} \left(\frac{\mu_i}{\mu_i + 1} \right)^{y_i} \left(\frac{1}{\mu_i + 1} \right)$$

Even though the parameters brms yielded look different from those in the text, their predictions describe the data well. Here's the `marginal_effects()` plot:

```
plot(marginal_effects(b10.18),
      points = T,
      point_args = c(size = 3/5, alpha = 2/3),
      line_args = c(color = wes_palette("Moonrise2")[1],
                    fill = wes_palette("Moonrise2")[1]))
```



Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
```

```
## [1] parallel stats      graphics grDevices utils      datasets methods   base
##
## other attached packages:
## [1] tidybayes_1.0.4        loo_2.1.0           broom_0.5.1        bayesplot_1.6.0
## [5] ggthemes_4.0.1         wesanderson_0.3.6.9000 forcats_0.3.0     stringr_1.4.0
## [9] dplyr_0.8.0.1          purrr_0.2.5          readr_1.1.1       tidyverse_1.2.1
## [13] tibble_2.1.1           tidyverse_1.2.1     brms_2.8.8        Rcpp_1.0.1
## [17] rstan_2.18.2           StanHeaders_2.18.0-1 ggplot2_3.1.1
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2        ggridges_0.5.0      rsconnect_0.8.8    rprojroot_1.3-2
## [5] ggstance_0.3            markdown_0.8         base64enc_0.1-3    rstudioapi_0.7
## [9] listenv_0.7.0           svUnit_0.7-12       DT_0.4             fansi_0.4.0
## [13] mvtnorm_1.0-10          lubridate_1.7.4     xml2_1.2.0        bridgesampling_0.6-0
## [17] codetools_0.2-15         mnormt_1.5-5        knitr_1.20         shinythemes_1.1.1
## [21] jsonlite_1.5            shiny_1.1.0          compiler_3.5.1    httr_1.3.1
## [25] backports_1.1.4         assertthat_0.2.0    Matrix_1.2-14     lazyeval_0.2.2
## [29] cli_1.0.1               later_0.7.3         htmltools_0.3.6   prettyunits_1.0.2
## [33] tools_3.5.1             igraph_1.2.1        coda_0.19-2       gtable_0.3.0
## [37] glue_1.3.1.9000          reshape2_1.4.3     cellranger_1.1.0 nlme_3.1-137
## [41] crosstalk_1.0.0          psych_1.8.4        xfun_0.3           globals_0.12.4
## [45] ps_1.2.1                rvest_0.3.2        mime_0.5           miniUI_0.1.1.1
## [49] gtools_3.8.1             future_1.12.0     MASS_7.3-50       zoo_1.8-2
## [53] scales_1.0.0             colourpicker_1.0   hms_0.4.2          promises_1.0.1
## [57] Brobdingnag_1.2-6        inline_0.3.15      shinystan_2.5.0   yaml_2.1.19
## [61] gridExtra_2.3             stringi_1.4.3      dygraphs_1.1.1.5  pkgbuild_1.0.2
## [65] rlang_0.3.4              pkgconfig_2.0.2    matrixStats_0.54.0 HDInterval_0.2.0
## [69] evaluate_0.10.1           lattice_0.20-35   rstantools_1.5.1 htmlwidgets_1.2
## [73] labeling_0.3              processx_3.2.1    tidyselect_0.2.5  plyr_1.8.4
## [77] magrittr_1.5              bookdown_0.9       R6_2.3.0           generics_0.0.2
## [81] foreign_0.8-70            pillar_1.3.1      haven_1.1.2       withr_2.1.2
## [85] xts_0.10-2               abind_1.4-5        modelr_0.1.2      crayon_1.3.4
## [89] arrayhelpers_1.0-20160527 utf8_1.1.4        rmarkdown_1.10     grid_3.5.1
## [93] readxl_1.1.0              callr_3.1.0       threejs_0.3.1     digest_0.6.18
## [97] xtable_1.8-2              httpuv_1.4.4.2    stats4_3.5.1      munsell_0.5.0
## [101] shinyjs_1.0
```

Chapter 11

Monsters and Mixtures

[Of these majestic creatures], we'll consider two common and useful examples. The first type is the ordered categorical model, useful for categorical outcomes with a fixed ordering. This model is built by merging a categorical likelihood function with a special kind of link function, usually a cumulative link. The second type is a family of zero-inflated and zero-augmented models, each of which mixes a binary event within an ordinary GLM likelihood like a Poisson or binomial.

Both types of models help us transform our modeling to cope with the inconvenient realities of measurement, rather than transforming measurements to cope with the constraints of our models. (p. 331)

11.1 Ordered categorical outcomes

It is very common in the social sciences, and occasional in the natural sciences, to have an outcome variable that is discrete, like a count, but in which the values merely indicate different ordered *levels* along some dimension. For example, if I were to ask you how much you like to eat fish, on a scale from 1 to 7, you might say 5. If I were to ask 100 people the same question, I'd end up with 100 values between 1 and 7. In modeling each outcome value, I'd have to keep in mind that these values are *ordered* because 7 is greater than 6, which is greater than 5, and so on. But unlike a count, the differences in values are not necessarily equal.

In principle, an ordered categorical variable is just a multinomial prediction problem (page 323). But the constraint that the categories be ordered demands special treatment...

The conventional solution is to use a cumulative link function. The cumulative probability of a value is the probability of that *value or any smaller value*. (pp. 331–332, *emphasis* in the original)

11.1.1 Example: Moral intuition.

Let's get the Trolley data from rethinking.

```
library(rethinking)

## Warning: package 'ggplot2' was built under R version 3.5.2

data(Trolley)
d <- Trolley
```

Unload rethinking and load brms.

```
rm(Trolley)
detach(package:rethinking, unload = T)
library(brms)

## Warning: package 'Rcpp' was built under R version 3.5.2
```

Use the tidyverse to get a sense of the dimensions of the data.

```
library(tidyverse)

glimpse(d)

## # Observations: 9,930
## # Variables: 12
## $ case      <fct> cfaqu, cfbur, cfrub, cibox, cibur, cispe, fkaqu, fkb...
## $ response   <int> 4, 3, 4, 3, 3, 3, 5, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4...
## $ order      <int> 2, 31, 16, 32, 4, 9, 29, 12, 23, 22, 27, 19, 14, 3, ...
## $ id         <fct> 96;434, 96;434, 96;434, 96;434, 96;434, 96;434, 96;4...
## $ age        <int> 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, ...
## $ male       <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ edu        <fct> Middle School, Middle School, Middle School, Middle ...
## $ action      <int> 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0...
## $ intention   <int> 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ contact     <int> 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
## $ story       <fct> aqu, bur, rub, box, bur, spe, aqu, boa, box, bur, ca...
## $ action2     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0...
```

Though we have 9,930 rows, we only have 331 unique individuals.

```
d %>%
  distinct(id) %>%
  count()

## # A tibble: 1 x 1
##       n
##   <int>
## 1    331
```

11.1.2 Describing an ordered distribution with intercepts.

Before we get to plotting, in this chapter we'll use theme settings and a color palette from the [ggthemes package](#).

```
library(ggthemes)
```

We'll take our basic theme settings from the `theme_hc()` function. We'll use the `Green fields` color palette, which we can inspect with the `canva_pal()` function and a little help from `scales::show_col()`.

```
scales::show_col(canva_pal("Green fields")(4))
```



```
canva_pal("Green fields")(4)

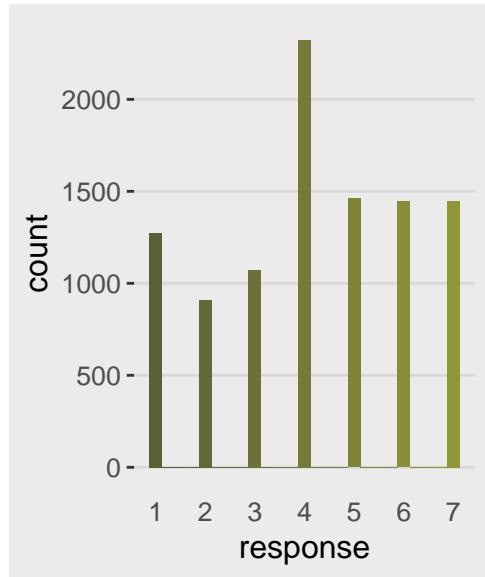
## [1] "#919636" "#524a3a" "#fffae1" "#5a5f37"

canva_pal("Green fields")(4)[3]

## [1] "#fffae1"
```

Now we're ready to make our ggplot2 version of the simple histogram, Figure 11.1.a.

```
ggplot(data = d, aes(x = response, fill = ..x..)) +
  geom_histogram(binwidth = 1/4, size = 0) +
  scale_x_continuous(breaks = 1:7) +
  scale_fill_gradient(low = canva_pal("Green fields")(4)[4],
                      high = canva_pal("Green fields")(4)[1]) +
  theme_hc() +
  theme(axis.ticks.x = element_blank(),
        plot.background = element_rect(fill = "grey92"),
        legend.position = "none")
```

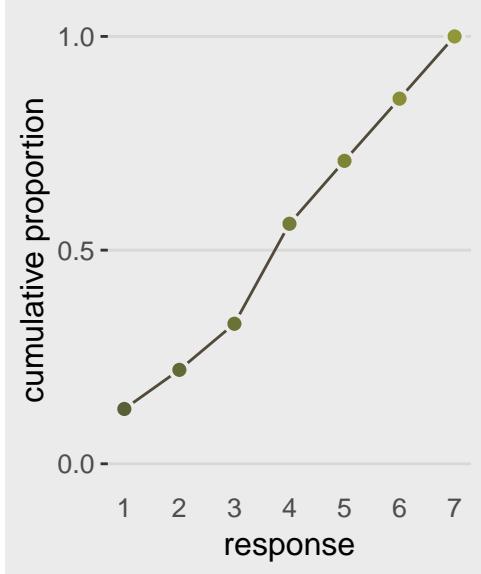


Our cumulative proportion plot, Figure 11.1.b, will require some pre-plot wrangling.

```
d %>%
  group_by(response) %>%
  count() %>%
  mutate(pr_k      = n / nrow(d)) %>%
  ungroup() %>%
  mutate(cum_pr_k = cumsum(pr_k)) %>%

  ggplot(aes(x = response, y = cum_pr_k,
             fill = response)) +
  geom_line(color = canva_pal("Green fields")(4)[2]) +
  geom_point(shape = 21, colour = "grey92",
             size = 2.5, stroke = 1) +
  scale_x_continuous(breaks = 1:7) +
  scale_y_continuous("cumulative proportion", breaks = c(0, .5, 1)) +
  scale_fill_gradient(low = canva_pal("Green fields")(4)[4],
                      high = canva_pal("Green fields")(4)[1]) +
  coord_cartesian(ylim = c(0, 1)) +
```

```
theme_hc() +
  theme(axis.ticks.x = element_blank(),
        plot.background = element_rect(fill = "grey92"),
        legend.position = "none")
```

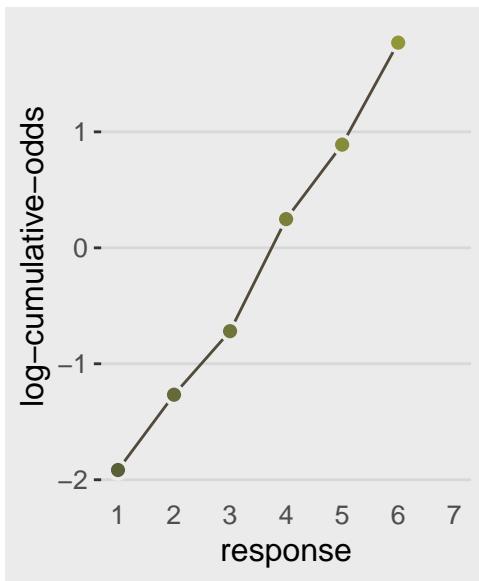


In order to make the next plot, we'll need McElreath's `logit()` function. Here it is, the logarithm of cumulative odds plot, Figure 11.1.c.

```
# McElreath's convenience function from page 335
logit <- function(x) log(x / (1 - x))

d %>%
  group_by(response) %>%
  count() %>%
  mutate(pr_k      = n / nrow(d)) %>%
  ungroup() %>%
  mutate(cum_pr_k = cumsum(pr_k)) %>%
  filter(response < 7) %>%

# we can do the `logit()` conversion right in ggplot2
ggplot(aes(x = response, y = logit(cum_pr_k),
            fill = response)) +
  geom_line(color = canva_pal("Green fields")(4)[2]) +
  geom_point(shape = 21, colour = "grey92",
             size = 2.5, stroke = 1) +
  scale_x_continuous(breaks = 1:7) +
  scale_fill_gradient(low = canva_pal("Green fields")(4)[4],
                      high = canva_pal("Green fields")(4)[1]) +
  coord_cartesian(xlim = c(1, 7)) +
  ylab("log-cumulative-odds") +
  theme_hc() +
  theme(axis.ticks.x = element_blank(),
        plot.background = element_rect(fill = "grey92"),
        legend.position = "none")
```

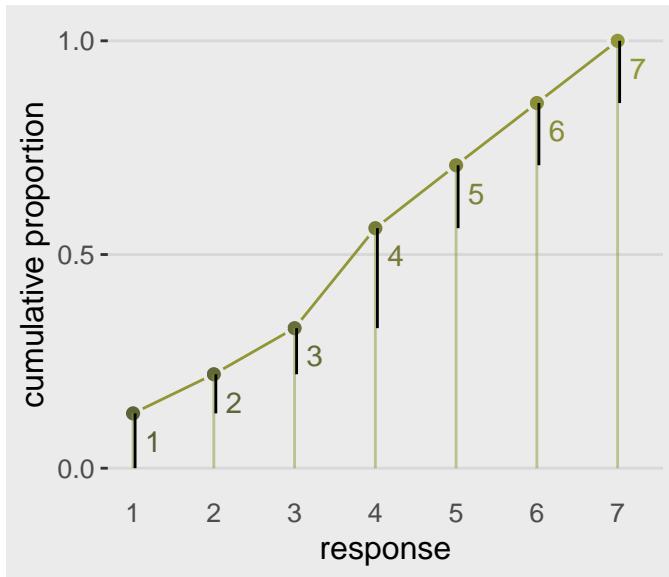


The code for Figure 11.2 is itself something of a monster.

```
d_plot <-
d %>%
group_by(response) %>%
count() %>%
mutate(pr_k      = n / nrow(d)) %>%
ungroup() %>%
mutate(cum_pr_k = cumsum(pr_k))

ggplot(data = d_plot,
       aes(x = response, y = cum_pr_k,
            color = cum_pr_k, fill = cum_pr_k)) +
  geom_line(color = canva_pal("Green fields")(4)[1]) +
  geom_point(shape = 21, colour = "grey92",
             size = 2.5, stroke = 1) +
  geom_linerange(aes(ymax = 0, ymin = cum_pr_k),
                 alpha = 1/2, color = canva_pal("Green fields")(4)[1]) +
# there must be more elegant ways to do this part
  geom_linerange(data = . %>%
                     mutate(discrete_probability =
                           ifelse(response == 1, cum_pr_k,
                                  cum_pr_k - pr_k)),
                 aes(x     = response + .025,
                     ymin = ifelse(response == 1, 0, discrete_probability),
                     ymax = cum_pr_k,
                     color = "black")) +
  geom_text(data = tibble(text      = 1:7,
                         response = seq(from = 1.25, to = 7.25, by = 1),
                         cum_pr_k = d_plot$cum_pr_k - .065),
            aes(label = text),
            size = 4) +
  scale_x_continuous(breaks = 1:7) +
  scale_y_continuous("cumulative proportion", breaks = c(0, .5, 1)) +
  scale_fill_gradient(low = canva_pal("Green fields")(4)[4],
                      high = canva_pal("Green fields")(4)[1]) +
  scale_color_gradient(low = canva_pal("Green fields")(4)[4],
                      high = canva_pal("Green fields")(4)[1]) +
  coord_cartesian(ylim = c(0, 1)) +
  theme_hc() +
  theme(axis.ticks.x     = element_blank(),
```

```
plot.background = element_rect(fill = "grey92"),
legend.position = "none")
```



McElreath's convention for this first type of statistical model is

$$\begin{aligned} R_i &\sim \text{Ordered}(\mathbf{p}) \\ \text{logit}(p_k) &= \alpha_k \\ \alpha_k &\sim \text{Normal}(0, 10) \end{aligned}$$

The Ordered distribution is really just a categorical distribution that takes a vector $\mathbf{p} = p_1, p_2, p_3, p_4, p_5, p_6$ of probabilities of each response value below the maximum response (7 in this example). Each response value k in this vector is defined by its link to an intercept parameter, α_k . Finally, some weakly regularizing priors are placed on these intercepts. (p. 335)

Whereas in `rethinking::map()` you indicate the likelihood by `<criterion> ~ dordlogit(phi, c(<the thresholds>))`, in `brms::brm()` you code `family = cumulative`. Here's the intercepts-only model:

```
# define the start values
inits <- list(`Intercept[1]` = -2,
             `Intercept[2]` = -1,
             `Intercept[3]` = 0,
             `Intercept[4]` = 1,
             `Intercept[5]` = 2,
             `Intercept[6]` = 2.5)

inits_list <- list(inits, inits)

b11.1 <-
  brm(data = d, family = cumulative,
       response ~ 1,
       prior(normal(0, 10), class = Intercept),
       iter = 2000, warmup = 1000, cores = 2, chains = 2,
       inits = inits_list, # here we add our start values
       seed = 11)
```

McElreath needed to include the `depth=2` argument in the `rethinking::precis()` function to show the threshold parameters from his `m11.1stan` model. With a `brm()` fit, we just use `print()` or `summary()` as usual.

```
print(b11.1)

## Family: cumulative
## Links: mu = logit; disc = identity
## Formula: response ~ 1
## Data: d (Number of observations: 9930)
## Samples: 2 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 2000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept[1] -1.92     0.03   -1.97    -1.86      1476 1.00
## Intercept[2] -1.27     0.02   -1.31    -1.22      1821 1.00
## Intercept[3] -0.72     0.02   -0.76    -0.68      2021 1.00
## Intercept[4]  0.25     0.02    0.21     0.29      2216 1.00
## Intercept[5]  0.89     0.02    0.85     0.93      2327 1.00
## Intercept[6]  1.77     0.03    1.72     1.83      2482 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

What McElreath's `m11.1stan` summary termed `cutpoints[k]`, ours termed `Intercept[k]`. In both cases, these are the α_k parameters from the equations, above. The summaries look like those in the text, number of effective samples are high, and the \hat{R} values are great. The model looks good.

Recall we use the `brms::inv_logit_scaled()` function in place of McElreath's `logistic()` function to get these into the probability metric.

```
b11.1 %>%
  fixef() %>%
  inv_logit_scaled()

##             Estimate Est.Error Q2.5     Q97.5
## Intercept[1] 0.1282430 0.5072421 0.1222287 0.1346667
## Intercept[2] 0.2197474 0.5058551 0.2120979 0.2277862
## Intercept[3] 0.3277541 0.5053708 0.3181448 0.3369461
## Intercept[4] 0.5616313 0.5051667 0.5515424 0.5715200
## Intercept[5] 0.7090137 0.5054678 0.7006442 0.7176885
## Intercept[6] 0.8545209 0.5071545 0.8475148 0.8613844
```

But recall that the posterior *SD* (i.e., the 'Est.Error' values) are not valid using that approach. If you really care about them, you'll need to work with the `posterior_samples()`.

```
posterior_samples(b11.1) %>%
  select(starts_with("b_")) %>%
  mutate_all(inv_logit_scaled) %>%
  gather() %>%
  group_by(key) %>%
  summarise(mean = mean(value),
            sd   = sd(value),
            ll   = quantile(value, probs = .025),
            ul   = quantile(value, probs = .975))

## # A tibble: 6 x 5
##   key           mean     sd     ll     ul
##   <chr>        <dbl>   <dbl> <dbl> <dbl>
## 1 b_Intercept[1] 0.128 0.00324 0.122 0.135
```

```
## 2 b_Intercept[2] 0.220 0.00402 0.212 0.228
## 3 b_Intercept[3] 0.328 0.00473 0.318 0.337
## 4 b_Intercept[4] 0.562 0.00509 0.552 0.572
## 5 b_Intercept[5] 0.709 0.00451 0.701 0.718
## 6 b_Intercept[6] 0.854 0.00356 0.848 0.861
```

11.1.3 Adding predictor variables.

Now we define the linear model as $\phi_i = \beta x_i$. Accordingly, the formula for our cumulative logit model becomes

$$\log \frac{\Pr(y_i \leq k)}{1 - \Pr(y_i \leq k)} = \alpha_k - \phi_i$$

$$\phi_i = \beta x_i$$

I'm not aware that brms has an equivalent to the `rethinking::dordlogit()` function. So here we'll make it by hand. The code comes from McElreath's [GitHub page](#).

```
# first, we needed to specify the `logistic()` function, which is apart of the `dordlogit()` function
logistic <- function(x) {
  p <- 1 / (1 + exp(-x))
  p <- ifelse(x == Inf, 1, p)
  p
}

# now we get down to it
dordlogit <-
  function(x, phi, a, log = FALSE) {
    a <- c(as.numeric(a), Inf)
    p <- logistic(a[x] - phi)
    na <- c(-Inf, a)
    np <- logistic(na[x] - phi)
    p <- p - np
    if (log == TRUE) p <- log(p)
    p
}
```

The `dordlogit()` function works like this:

```
(pk <- dordlogit(1:7, 0, fixef(b11.1)[, 1]))
```

```
## [1] 0.12824305 0.09150433 0.10800668 0.23387727 0.14738234 0.14550720
## [7] 0.14547915
```

Note the slight difference in how we used `dordlogit()` with a `brm()` fit summarized by `fixef()` than the way McElreath did with a `map2stan()` fit summarized by `coef()`. McElreath just put `coef(m11.1)` into `dordlogit()`. We, however, more specifically placed `fixef(b11.1)[, 1]` into the function. With the `[, 1]` part, we specified that we were working with the posterior means (i.e., `Estimate`) and neglecting the other summaries (i.e., the posterior *SDs* and 95% intervals). If you forget to subset, chaos ensues.

Next, as McElreath further noted in the text, “these probabilities imply an average outcome of:”

```
sum(pk * (1:7))
```

```
## [1] 4.19909
```

I found that a bit abstract. Here's the thing in a more elaborate tibble format.

```

(
  explicit_example <-
  tibble(probability_of_a_response = pk) %>%
  mutate(the_response = 1:7) %>%
  mutate(their_product = probability_of_a_response * the_response)
)

## # A tibble: 7 x 3
##   probability_of_a_response the_response their_product
##   <dbl>           <int>          <dbl>
## 1 0.128             1      0.128
## 2 0.0915            2      0.183
## 3 0.108             3      0.324
## 4 0.234             4      0.936
## 5 0.147             5      0.737
## 6 0.146             6      0.873
## 7 0.145             7      1.02

explicit_example %>%
  summarise(average_outcome_value = sum(their_product))

```

```

## # A tibble: 1 x 1
##   average_outcome_value
##   <dbl>
## 1 4.20

```

Aside

This made me wonder how this would compare if we were lazy and ignored the categorical nature of the `response`. Here we refit the model with the typical Gaussian likelihood.

```

brm(data = d, family = gaussian,
  response ~ 1,
  # in this case, 4 (i.e., the middle response) seems to be the conservative place to put the mean
  prior = c(prior(normal(4, 10), class = Intercept),
            prior(cauchy(0, 1), class = sigma)),
  iter = 2000, warmup = 1000, cores = 4, chains = 4,
  seed = 11) %>%
  print()

```

```

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: response ~ 1
## Data: d (Number of observations: 9930)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     4.20      0.02     4.16     4.24        2600 1.00
## 
## Family Specific Parameters:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.91      0.01     1.88     1.93        3102 1.00
## 
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Happily, this yielded a mean estimate of 4.2, much like our `average_outcome_value`, above.

End aside

Now we'll try it by subtracting .5 from each.

```
# the probabilities of a given response
(pk <- dordlogit(1:7, 0, fixef(b11.1)[, 1] - .5))

## [1] 0.08191684 0.06398163 0.08232659 0.20905342 0.15914829 0.18440296
## [7] 0.21917028

# the average rating
sum(pk * (1:7))

## [1] 4.729425
```

So the rule is we *subtract the linear model from each intercept*. Let's fit our multivariable models.

```
# start values for b11.2
inits <- list(`Intercept[1]` = -1.9,
  `Intercept[2]` = -1.2,
  `Intercept[3]` = -0.7,
  `Intercept[4]` = 0.2,
  `Intercept[5]` = 0.9,
  `Intercept[6]` = 1.8,
  action = 0,
  intention = 0,
  contact = 0)

b11.2 <-
  brm(data = d, family = cumulative,
    response ~ 1 + action + intention + contact,
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(normal(0, 10), class = b)),
    iter = 2000, warmup = 1000, cores = 2, chains = 2,
    inits = list(inits, inits),
    seed = 11)

# start values for b11.3
inits <- list(`Intercept[1]` = -1.9,
  `Intercept[2]` = -1.2,
  `Intercept[3]` = -0.7,
  `Intercept[4]` = 0.2,
  `Intercept[5]` = 0.9,
  `Intercept[6]` = 1.8,
  action = 0,
  intention = 0,
  contact = 0,
  `action:intention` = 0,
  `contact:intention` = 0)

b11.3 <-
  update(b11.2,
    formula = response ~ 1 + action + intention + contact + action:intention + contact:intention,
    inits = list(inits, inits))
```

We don't have a `coeftab()` function in `brms` like for `rethinking`. But as we did for Chapter 6, we can reproduce it with help from the `broom package` and a bit of data wrangling.

```
library(broom)

tibble(model = str_c("b11.", 1:3)) %>%
  mutate(fit = purrr::map(model, get)) %>%
  mutate(tidy = purrr::map(fit, tidy)) %>%
  unnest(tidy) %>%
  select(model, term, estimate) %>%
  filter(term != "lp__") %>%
  complete(term = distinct(., term), model) %>%
  mutate(estimate = round(estimate, digits = 2)) %>%
  spread(key = model, value = estimate) %>%
  # this last step isn't necessary, but it orders the rows to match the text
  slice(c(6:11, 1, 4, 3, 2, 5))
```

```
## # A tibble: 11 x 4
##   term          b11.1   b11.2   b11.3
##   <chr>        <dbl>    <dbl>    <dbl>
## 1 b_Intercept [1] -1.92   -2.84   -2.64
## 2 b_Intercept [2] -1.27   -2.16   -1.94
## 3 b_Intercept [3] -0.72   -1.57   -1.35
## 4 b_Intercept [4]  0.25   -0.55   -0.31
## 5 b_Intercept [5]  0.89    0.12    0.36
## 6 b_Intercept [6]  1.77    1.02    1.27
## 7 b_action       NA     -0.71   -0.47
## 8 b_intention    NA     -0.72   -0.290
## 9 b_contact      NA     -0.96   -0.33
## 10 b_action:intention NA     NA     -0.44
## 11 b_intention:contact NA     NA     -1.27
```

If you really wanted that last `nobs` row at the bottom, you could elaborate on this code: `b11.1$data %>% count()`. Also, if you want a proper `coeftab()` function for brms, McElreath's code lives [here](#). Give it a whirl.

Here we compute the WAIC. *Caution: This took some time to compute.*

```
b11.1 <- add_criterion(b11.1, "waic")
b11.2 <- add_criterion(b11.2, "waic")
b11.3 <- add_criterion(b11.3, "waic")
```

Now compare the models.

```
loo_compare(b11.1, b11.2, b11.3, criterion = "waic") %>%
  print(simplify = F)
```

```
##           elpd_diff se_diff elpd_waic se_elpd_waic p_waic    se_p_waic
## b11.3      0.0     0.0 -18464.6     40.6      11.0     0.1
## b11.2     -80.4    12.8 -18544.9     38.1      9.0      0.0
## b11.1    -462.6    31.3 -18927.2     28.8      6.0      0.0
##           waic    se_waic
## b11.3  36929.1    81.2
## b11.2  37089.9    76.3
## b11.1  37854.4    57.6
```

Here are the WAIC weights.

```
model_weights(b11.1, b11.2, b11.3,
              weights = "waic") %>%
  round(digits = 3)
```

```
## b11.1 b11.2 b11.3
##      0      0      1
```

McElreath made Figure 11.3 by extracting the samples of his `m11.3`, saving them as `post`, and working some hairy base R `plot()` code. We'll take a different route and use `brms::fitted()`. This will take substantial data wrangling, but hopefully it'll be instructive. Let's first take a look at the initial `fitted()` output for the beginnings of Figure 11.3.a.

```
nd <-
  tibble(action    = 0,
         contact   = 0,
         intention = 0:1)

max_iter <- 100

fitted(b11.3,
       newdata = nd,
       subset  = 1:max_iter,
       summary = F) %>%
  as_tibble() %>%
  glimpse()

## #> #> Observations: 100
## #> #> Variables: 14
## #> #> $ `1.1` <dbl> 0.06528924, 0.06587534, 0.06325178, 0.06988218, 0.067933...
## #> #> $ `2.1` <dbl> 0.08835360, 0.07956653, 0.09421488, 0.08616458, 0.089860...
## #> #> $ `1.2` <dbl> 0.05481430, 0.06596464, 0.05105144, 0.06284758, 0.054994...
## #> #> $ `2.2` <dbl> 0.07087948, 0.07737109, 0.07161809, 0.07495434, 0.069705...
## #> #> $ `1.3` <dbl> 0.08031083, 0.08192224, 0.07743267, 0.07891706, 0.081704...
## #> #> $ `2.3` <dbl> 0.09880285, 0.09302711, 0.10179266, 0.09089161, 0.098880...
## #> #> $ `1.4` <dbl> 0.2114573, 0.2228572, 0.2058134, 0.2136085, 0.2205930, 0...
## #> #> $ `2.4` <dbl> 0.2347824, 0.2372134, 0.2364694, 0.2294662, 0.2420963, 0...
## #> #> $ `1.5` <dbl> 0.1683519, 0.1573705, 0.1666625, 0.1632613, 0.1646739, 0...
## #> #> $ `2.5` <dbl> 0.1644616, 0.1548273, 0.1619477, 0.1607279, 0.1603060, 0...
## #> #> $ `1.6` <dbl> 0.1962602, 0.1930833, 0.1994937, 0.1951076, 0.1910063, 0...
## #> #> $ `2.6` <dbl> 0.1709020, 0.1771991, 0.1666971, 0.1774519, 0.1675730, 0...
## #> #> $ `1.7` <dbl> 0.2235162, 0.2129268, 0.2362946, 0.2163758, 0.2190943, 0...
## #> #> $ `2.7` <dbl> 0.1718181, 0.1807955, 0.1672602, 0.1803434, 0.1715787, 0...
```

Hopefully by now it's clear why we needed the `nd` tibble, which we made use of in the `newdata = nd` argument. Because we set `summary = F`, we get draws from the posterior instead of summaries. With `max_iter`, we controlled how many of those posterior draws we wanted. McElreath used 100, which he indicated at the top of page 341, so we followed suit. It took me a minute to wrap my head around the meaning of the 14 vectors, which were named by `brms::fitted()` default. Notice how each column is named by two numerals, separated by a period. That first numeral indicates which if the two `intention` values the draw is based on (i.e., 1 stands for `intention == 0`, 2 stands for `intention == 1`). The numbers on the right of the decimals are the seven response options for `response`. For each posterior draw, you get one of those for each value of `intention`. Finally, it might not be immediately apparent, but the values are in the probability scale, just like `pk` on page 338.

Now we know what we have in hand, it's just a matter of careful wrangling to get those probabilities into a more useful format to feed into `ggplot2`. I've extensively annotated the code, below. If you lose track of happens in a given step, just run the code up till that point. Go step by step.

```
nd <-
  tibble(action    = 0,
         contact   = 0,
         intention = 0:1)

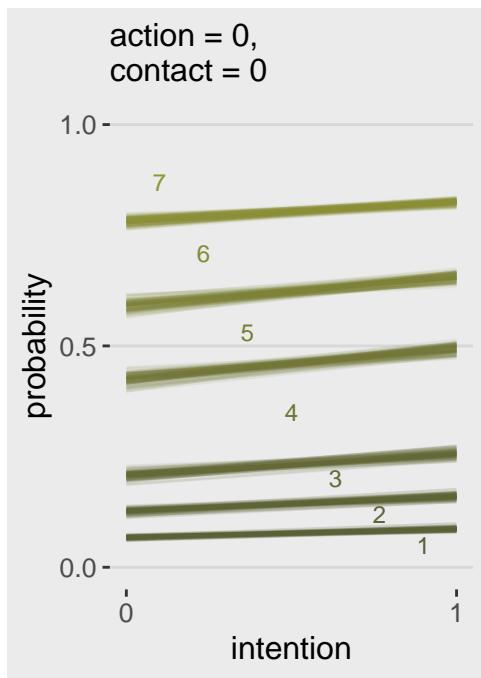
max_iter <- 100
```

```

fitted(b11.3,
       newdata = nd,
       subset   = 1:max_iter,
       summary  = F) %>%
as_tibble() %>%
# we convert the data to the long format
gather() %>%
# we need an variable to index which posterior iteration we're working with
mutate(iter = rep(1:max_iter, times = 14)) %>%
# this step isn't technically necessary, but I prefer my `iter` index at the far left.
select(iter, everything()) %>%
# here we extract the `intention` and `response` information out of the `key` vector and
# spread it into two vectors.
separate(key, into = c("intention", "rating")) %>%
# that step produced two character vectors. they'll be more useful as numbers
mutate(intention = intention %>% as.double(),
       rating    = rating %>% as.double()) %>%
# here we convert `intention` into its proper 0:1 metric
mutate(intention = intention -1) %>%
# this isn't necessary, but it helps me understand exactly what metric the values are currently in
rename(pk = value) %>%
# this step is based on McElreath's R code 11.10 on page 338
mutate(`pk:rating` = pk * rating) %>%
# I'm not sure how to succinctly explain this. you're just going to have to trust me
group_by(iter, intention) %>%
# this is very important for the next step.
arrange(iter, intention, rating) %>%
# here we take our `pk` values and make culmulative sums. why? take a long hard look at Figure 11.2.
mutate(probability = cumsum(pk)) %>%
# `rating == 7` is unnecessary. these `probability` values are by definition 1
filter(rating < 7) %>%

ggplot(aes(x = intention,
            y = probability,
            color = probability)) +
geom_line(aes(group = interaction(iter, rating)),
          alpha = 1/10) +
# note how we made a new data object for `geom_text()`
geom_text(data = tibble(text      = 1:7,
                        intention = seq(from = .9, to = .1, length.out = 7),
                        probability = c(.05, .12, .20, .35, .53, .71, .87)),
          aes(label = text),
          size = 3) +
scale_x_continuous(breaks = 0:1) +
scale_y_continuous(breaks = c(0, .5, 1)) +
coord_cartesian(ylim = 0:1) +
scale_color_gradient(low = canva_pal("Green fields")(4)[4],
                     high = canva_pal("Green fields")(4)[1]) +
labs(subtitle = "action = 0,\ncontact = 0",
     x        = "intention") +
theme_hc() +
theme(plot.background = element_rect(fill = "grey92"),
      legend.position = "none")

```



Boom!

Okay, that pile of code is a bit of a mess and you're not going to want to repeatedly cut and paste all that. Let's condense it into a homemade function, `make_Figure_11.3_data()`.

```
make_Figure_11.3_data <- function(action, contact, max_iter){

  nd <-
    tibble(action      = action,
           contact     = contact,
           intention = 0:1)

  max_iter <- max_iter

  fitted(b11.3,
         newdata = nd,
         subset  = 1:max_iter,
         summary = F) %>%
    as_tibble() %>%
    gather() %>%
    mutate(iter = rep(1:max_iter, times = 14)) %>%
    select(iter, everything()) %>%
    separate(key, into = c("intention", "rating")) %>%
    mutate(intention = intention %>% as.double(),
           rating    = rating %>% as.double()) %>%
    mutate(intention = intention -1) %>%
    rename(pk = value) %>%
    mutate(`pk:rating` = pk * rating) %>%
    group_by(iter, intention) %>%
    arrange(iter, intention, rating) %>%
    mutate(probability = cumsum(pk)) %>%
    filter(rating < 7)
}
```

Now we'll use our sweet homemade function to make our plots.

```
# Figure 11.3.a
p1 <-
```

```

make_Figure_11.3_data(action    = 0,
                      contact   = 0,
                      max_iter = 100) %>%

ggplot(aes(x = intention,
            y = probability,
            color = probability)) +
  geom_line(aes(group = interaction(iter, rating)),
            alpha = 1/10) +
  geom_text(data = tibble(text      = 1:7,
                          intention = seq(from = .9, to = .1, length.out = 7),
                          probability = c(.05, .12, .20, .35, .53, .71, .87)),
            aes(label = text),
            size = 3) +
  scale_x_continuous(breaks = 0:1) +
  scale_y_continuous(breaks = c(0, .5, 1)) +
  scale_color_gradient(low = canva_pal("Green fields")(4)[4],
                        high = canva_pal("Green fields")(4)[1]) +
  coord_cartesian(ylim = 0:1) +
  labs(subtitle = "action = 0,\ncontact = 0",
       x         = "intention") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        legend.position = "none")

# Figure 11.3.b
p2 <-
  make_Figure_11.3_data(action    = 1,
                        contact   = 0,
                        max_iter = 100) %>%

  ggplot(aes(x = intention,
              y = probability,
              color = probability)) +
    geom_line(aes(group = interaction(iter, rating)),
              alpha = 1/10) +
    geom_text(data = tibble(text      = 1:7,
                            intention = seq(from = .9, to = .1, length.out = 7),
                            probability = c(.12, .24, .35, .50, .68, .80, .92)),
              aes(label = text),
              size = 3) +
    scale_x_continuous(breaks = 0:1) +
    scale_y_continuous(breaks = c(0, .5, 1)) +
    scale_color_gradient(low = canva_pal("Green fields")(4)[4],
                          high = canva_pal("Green fields")(4)[1]) +
    coord_cartesian(ylim = 0:1) +
    labs(subtitle = "action = 1,\ncontact = 0",
         x         = "intention") +
    theme_hc() +
    theme(plot.background = element_rect(fill = "grey92"),
          legend.position = "none")

# Figure 11.3.c
p3 <-
  make_Figure_11.3_data(action    = 0,
                        contact   = 1,
                        max_iter = 100) %>%

  ggplot(aes(x = intention,

```

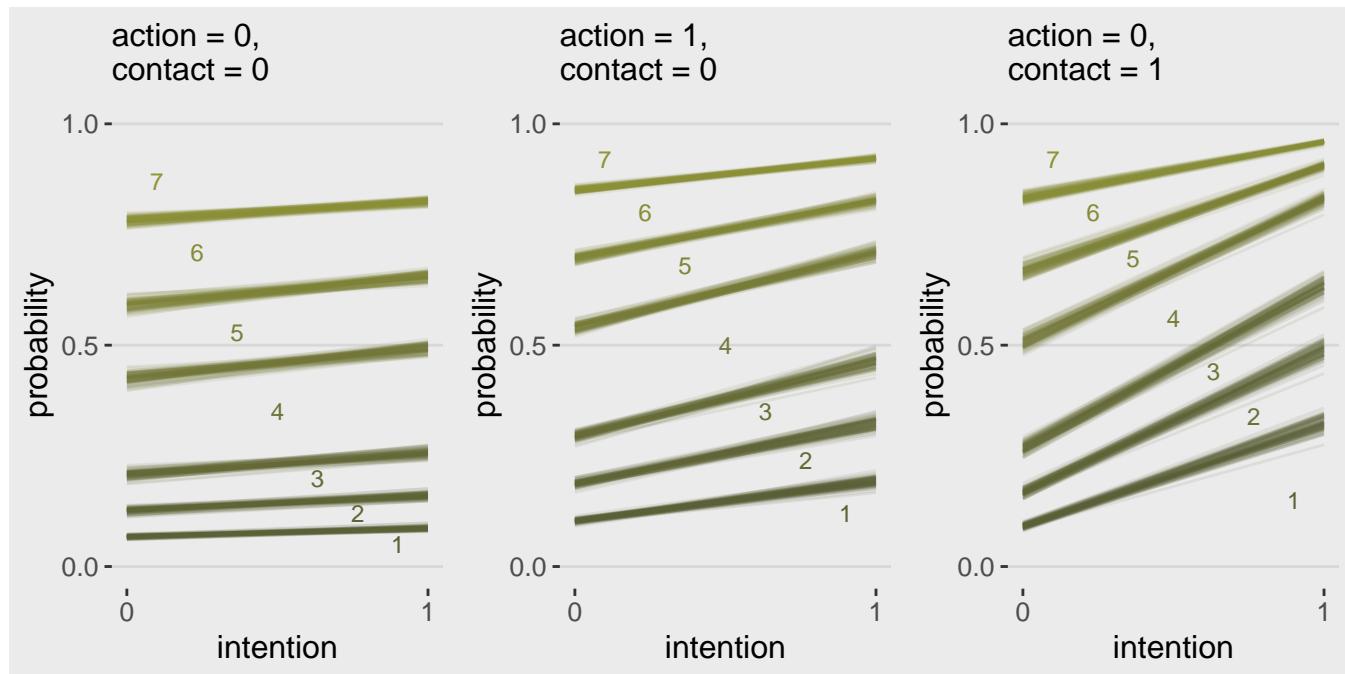
```

y = probability,
color = probability)) +
geom_line(aes(group = interaction(iter, rating)),
alpha = 1/10) +
geom_text(data = tibble(text      = 1:7,
intention  = seq(from = .9, to = .1, length.out = 7),
probability = c(.15, .34, .44, .56, .695, .8, .92)),
aes(label = text),
size = 3) +
scale_x_continuous(breaks = 0:1) +
scale_y_continuous(breaks = c(0, .5, 1)) +
scale_color_gradient(low = canva_pal("Green fields")(4)[4],
high = canva_pal("Green fields")(4)[1]) +
coord_cartesian(ylim = 0:1) +
labs(subtitle = "action = 0, \ncontact = 1",
x          = "intention") +
theme_hc() +
theme(plot.background = element_rect(fill = "grey92"),
legend.position = "none")

# here we stitch them together with `grid.arrange()`
library(gridExtra)

grid.arrange(p1, p2, p3, ncol = 3)

```



If you'd like to learn more about these kinds of models and how to fit them in brms, check out Bürkner and Vuorre's [Ordinal Regression Models in Psychology: A Tutorial](#).

11.1.4 Bonus: Figure 11.3 alternative.

I have a lot of respect for McElreath. But man, Figure 11.3 is the worst. I'm in clinical psychology and there's no way a working therapist is going to look at a figure like that and have any sense of what's going on. Nobody's got time for that. We've have clients to serve! Happily, we can go further. Look back at McElreath's R code 11.10 on page 338. See how he multiplied the elements of `pk` by their respective `response` values and then just summed them up to get an average outcome value? With just a little amendment to our custom `make_Figure_11.3_data()` function, we can wrangle our `fitted()` output to express average `response` values for each of our conditions of interest. Here's the adjusted function:

```

make_data_for_an_alternative_fiture <- function(action, contact, max_iter){

  nd <-
    tibble(action      = action,
           contact     = contact,
           intention = 0:1)

  max_iter <- max_iter

  fitted(b11.3,
         newdata = nd,
         subset  = 1:max_iter,
         summary = F) %>%
    as_tibble() %>%
    gather() %>%
    mutate(iter = rep(1:max_iter, times = 14)) %>%
    select(iter, everything()) %>%
    separate(key, into = c("intention", "rating")) %>%
    mutate(intention = intention %>% as.double(),
           rating   = rating %>% as.double()) %>%
    mutate(intention = intention -1) %>%
    rename(pk = value) %>%
    mutate(`pk:rating` = pk * rating) %>%
    group_by(iter, intention) %>%

  # everything above this point is identical to the previous custom function.
  # all we do is replace the last few lines with this one line of code.
  summarise(mean_rating = sum(`pk:rating`))
}

}

```

Our handy homemade but monstrously-named `make_data_for_an_alternative_fiture()` function works very much like its predecessor. You'll see.

```

# alternative to Figure 11.3.a
p1 <-
  make_data_for_an_alternative_fiture(action    = 0,
                                         contact   = 0,
                                         max_iter = 100) %>%

  ggplot(aes(x = intention, y = mean_rating, group = iter)) +
  geom_line(alpha = 1/10, color = canva_pal("Green fields")(4)[1]) +
  scale_x_continuous("intention", breaks = 0:1) +
  scale_y_continuous("response", breaks = 1:7) +
  coord_cartesian(ylim = 1:7) +
  labs(subtitle = "action = 0,\ncontact = 0") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        legend.position = "none")

# alternative to Figure 11.3.b
p2 <-
  make_data_for_an_alternative_fiture(action    = 1,
                                         contact   = 0,
                                         max_iter = 100) %>%

  ggplot(aes(x = intention, y = mean_rating, group = iter)) +
  geom_line(alpha = 1/10, color = canva_pal("Green fields")(4)[1]) +
  scale_x_continuous("intention", breaks = 0:1) +
  scale_y_continuous("response", breaks = 1:7) +

```

```

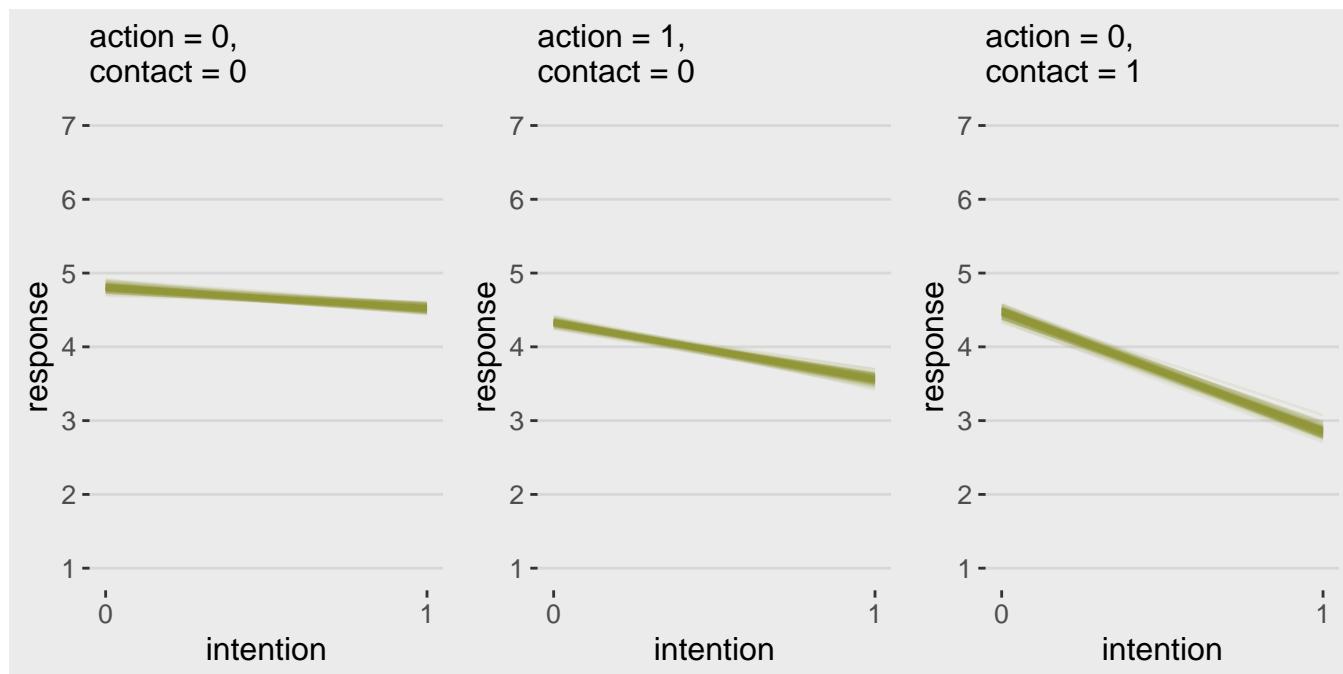
coord_cartesian(ylim = 1:7) +
labs(subtitle = "action = 1,\ncontact = 0") +
theme_hc() +
theme(plot.background = element_rect(fill = "grey92"),
legend.position = "none")

# alternative to Figure 11.3.c
p3 <-
make_data_for_an_alternative_fixture(action    = 0,
                                      contact   = 1,
                                      max_iter = 100) %>%

ggplot(aes(x = intention, y = mean_rating, group = iter)) +
geom_line(alpha = 1/10, color = canva_pal("Green fields")(4)[1]) +
scale_x_continuous("intention", breaks = 0:1) +
scale_y_continuous("response", breaks = 1:7) +
coord_cartesian(ylim = 1:7) +
labs(subtitle = "action = 0,\ncontact = 1") +
theme_hc() +
theme(plot.background = element_rect(fill = "grey92"),
legend.position = "none")

grid.arrange(p1, p2, p3, ncol = 3)

```



Finally; now those are plots I can sell in a clinical psychology journal!

11.2 Zero-inflated outcomes

Very often, the things we can measure are not emissions from any pure process. Instead, they are *mixtures* of multiple processes. Whenever there are different causes for the same observation, then a mixture model may be useful. A mixture model uses more than one simple probability distribution to model a mixture of causes. In effect, these models use more than one likelihood for the same outcome variable.

Count variables are especially prone to needing a mixture treatment. The reason is that a count of zero can often arise more than one way. A “zero” means that nothing happened, and nothing can happen either because the rate of events is low or rather because the process that generates events failed to get started. (p. 342, *emphasis* in the original)

In his **Rethinking: Breaking the law** box, McElreath discussed how advances in computing have made it possible for working scientists to define their own data generating models. If you'd like to dive deeper into the topic, check out Bürkner's vignette, [Define Custom Response Distributions with brms](#). We'll even make use of it a little further down.

11.2.1 Example: Zero-inflated Poisson.

Here we simulate our drunk monk data.

```
# define parameters
prob_drink <- 0.2 # 20% of days
rate_work <- 1      # average 1 manuscript per day

# sample one year of production
n <- 365

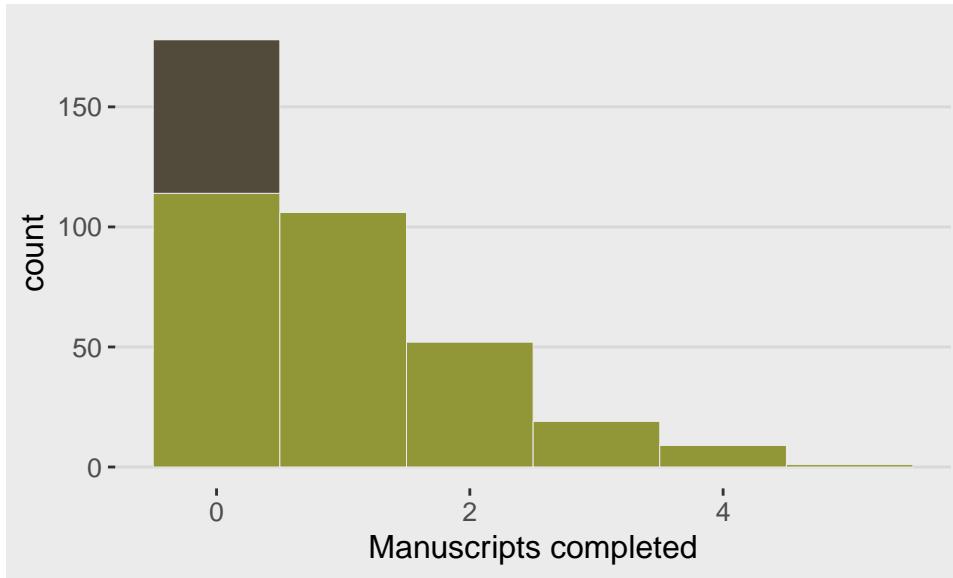
# simulate days monks drink
set.seed(11)
drink <- rbinom(n, 1, prob_drink)

# simulate manuscripts completed
y <- (1 - drink) * rpois(n, rate_work)
```

We'll put those data in a tidy tibble before plotting.

```
d <-
tibble(Y = y) %>%
arrange(Y) %>%
mutate(zeros = c(rep("zeros_drink", times = sum(drink)),
                rep("zeros_work",   times = sum(y == 0 & drink == 0)),
                rep("nope",          times = n - sum(y == 0))
              ))

ggplot(data = d, aes(x = Y)) +
  geom_histogram(aes(fill = zeros),
                 binwidth = 1, size = 1/10, color = "grey92") +
  scale_fill_manual(values = c(canva_pal("Green fields")(4)[1],
                               canva_pal("Green fields")(4)[2],
                               canva_pal("Green fields")(4)[1])) +
  xlab("Manuscripts completed") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        legend.position = "none")
```



With these data, the likelihood of observing zero on y , (i.e., the likelihood zero manuscripts were completed on a given occasion) is

$$\begin{aligned}\Pr(0|p, \lambda) &= \Pr(\text{drink}|p) + \Pr(\text{work}|p) \times \Pr(0|\lambda) \\ &= p + (1 - p) \exp(-\lambda)\end{aligned}$$

And

since the Poisson likelihood of y is $\Pr(y|\lambda) = \lambda^y \exp(-\lambda)/y!$, the likelihood of $y = 0$ is just $\exp(-\lambda)$. The above is just the mathematics for:

The probability of observing a zero is the probability that the monks didn't drink OR (+) the probability that the monks worked AND (\times) failed to finish anything.

And the likelihood of a non-zero value y is:

$$\begin{aligned}\Pr(y|p, \lambda) &= \Pr(\text{drink}|p)(0) + \Pr(\text{work}|p)\Pr(y|\lambda) \\ &= (1 - p) \frac{\lambda^y \exp(-\lambda)}{y!}\end{aligned}$$

Since drinking monks never produce $y > 0$, the expression above is just the chance the monks both work $1 - p$, and finish y manuscripts. (p. 344, *emphasis* in the original)

So letting p be the probability y is zero and λ be the shape of the distribution, the zero-inflated Poisson (ZIPoisson) regression model takes the basic form

$$\begin{aligned}y_i &\sim \text{ZIPoisson}(p_i, \lambda_i) \\ \text{logit}(p_i) &= \alpha_p + \beta_p x_i \\ \log(\lambda_i) &= \alpha_\lambda + \beta_\lambda x_i\end{aligned}$$

One last thing to note is that in brms, p_i is denoted `zi`. So the intercept [and `zi`] only zero-inflated Poisson model in brms looks like this.

```
b11.4 <-
  brm(data = d, family = zero_inflated_poisson,
    Y ~ 1,
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(beta(2, 2), class = zi)), # the brms default is beta(1, 1)
    cores = 4,
    seed = 11)

print(b11.4)

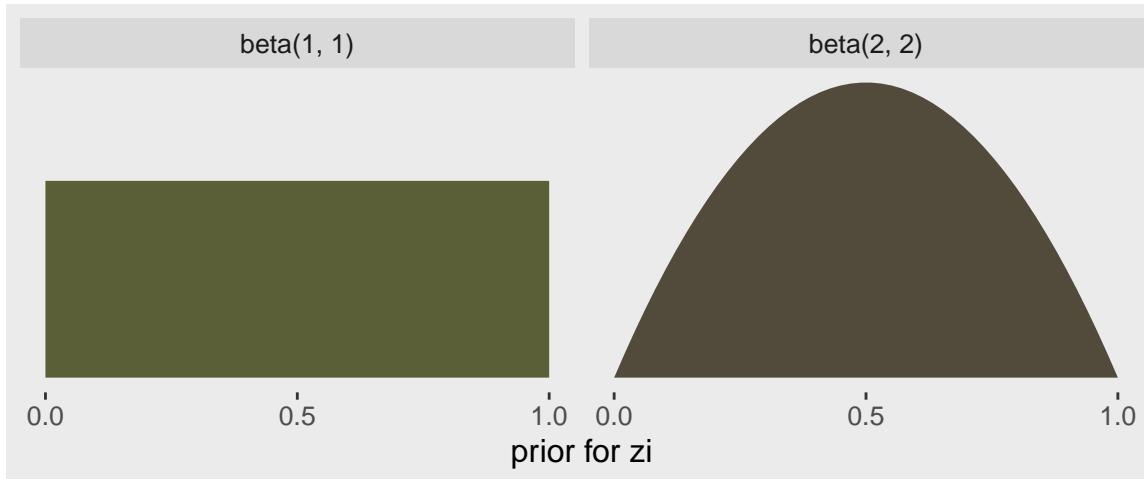
## Family: zero_inflated_poisson
##   Links: mu = log; zi = identity
## Formula: Y ~ 1
## Data: d (Number of observations: 365)
## Samples: 4 chains, each with iter = 2000; warmup = 1000; thin = 1;
##           total post-warmup samples = 4000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept     0.10      0.08     -0.08     0.26        1179 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## zi          0.24      0.05     0.13     0.33        1354 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

The zero-inflated Poisson is [parameterized in brms](#) a little differently than it is in rethinking. The different parameterization did not influence the estimate for the Intercept, λ . In both here and in the text, λ was about zero. However, it did influence the summary of `zi`. Note how McElreath's `logistic(-1.39)` yielded 0.1994078. Seems rather close to our `zi` estimate of 0.235. First off, because he didn't set his seed in the text before simulating, we couldn't exactly reproduce his simulated drunk monk data. So our results will vary a little due to that alone. But after accounting for simulation variance, hopefully it's clear that `zi` in brms is already in the probability metric. There's no need to convert it.

In the `prior` argument, we used `beta(2, 2)` for `zi` and also mentioned in the margin that the brms default is `beta(1, 1)`. To give you a sense of the priors, let's plot them.

```
tibble(`zi prior` = seq(from = 0, to = 1, length.out = 50)) %>%
  mutate(`beta(1, 1)` = dbeta(`zi prior`, 1, 1),
        `beta(2, 2)` = dbeta(`zi prior`, 2, 2)) %>%
  gather(prior, density, -`zi prior`) %>%

ggplot(aes(x = `zi prior`,
           ymin = 0,
           ymax = density)) +
  geom_ribbon(aes(fill = prior)) +
  scale_fill_manual(values = c(canva_pal("Green fields")(4)[4],
                               canva_pal("Green fields")(4)[2])) +
  scale_x_continuous("prior for zi", breaks = c(0, .5, 1)) +
  scale_y_continuous(NULL, breaks = NULL) +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        legend.position = "none") +
  facet_wrap(~prior)
```



Hopefully this clarifies that the brms default is flat, whereas our prior regularized a bit toward .5. Anyway, here's that exponentiated λ .

```
fixef(b11.4)[1, ] %>%
  exp()
```

```
## Estimate Est.Error Q2.5 Q97.5
## 1.0998806 1.0881350 0.9271772 1.2984891
```

11.2.1.1 Overthinking: Zero-inflated Poisson distribution function.

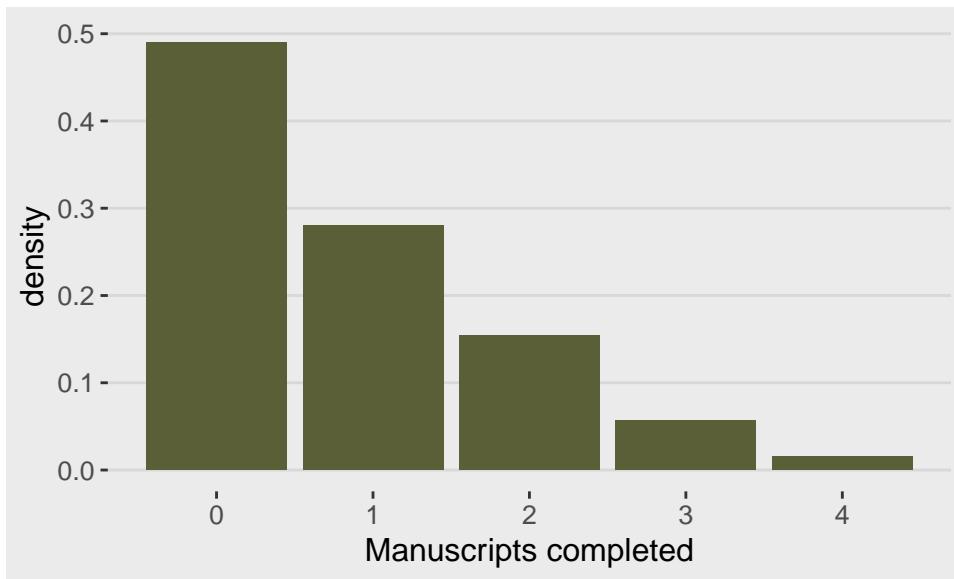
```
dzip <- function(x, p, lambda, log = TRUE) {
  ll <- ifelse(
    x == 0,
    p + (1 - p) * exp(-lambda),
    (1 - p) * dpois(x, lambda, log = FALSE)
  )
  if (log == TRUE) ll <- log(ll)
  return(ll)
}
```

We can use McElreath's `dzip()` to do a posterior predictive check for our model. To work with our estimates for p and λ directly, we'll set `log = F`.

```
p_b11.4      <- posterior_summary(b11.4)[2, 1]
lambda_b11.4 <- posterior_summary(b11.4)[1, 1] %>% exp()

tibble(x = 0:4) %>%
  mutate(density = dzip(x = x,
                        p = p_b11.4,
                        lambda = lambda_b11.4,
                        log = F)) %>%

ggplot(aes(x = x, y = density)) +
  geom_col(fill = canva_pal("Green fields")(4)[4]) +
  xlab("Manuscripts completed") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))
```



If you look up to the histogram we made at the beginning of this section, you'll see this isn't a terrible approximation.

11.3 Over-dispersed outcomes

All statistical models omit something. The question is only whether that something is necessary for making useful inferences. One symptom that something important has been omitted from a count model is over-dispersion. The variance of a variable is sometimes called its *dispersion*. For a counting process like a binomial, the variance is a function of the same parameters as the expected value. For example, the expected value of a binomial is np and its variance is $np(1-p)$. When the observed variance exceeds this amount—after conditioning on all the predictor variables—this implies that some omitted variable is producing additional dispersion in the observed counts.

What could go wrong, if we ignore the over-dispersion? Ignoring it can lead to all of the same problems as ignoring any predictor variable. Heterogeneity in counts can be a confound, hiding effects of interest or producing spurious inferences. (p, 346, *emphasis* in the original)

In this chapter we'll cope with the problem using continuous mixture models—first the beta-binomial and then the gamma-Poisson (a.k.a. negative binomial).

11.3.1 Beta-binomial.

A beta-binomial model assumes that each binomial count observation has its own probability of success. The model estimates the *distribution* of probabilities of success across cases, instead of a single probability of success. And predictor variables change the shape of this distribution, instead of directly determining the probability of each success. (p, 347, *emphasis* in the original)

Unfortunately, we need to digress. As it turns out, there are multiple ways to parameterize the beta distribution and we've run square into two. In the text, McElreath wrote the beta distribution has two parameters, an average probability \bar{p} and a shape parameter θ . In his R code 11.24, which we'll reproduce in a bit, he demonstrated that parameterization with the `rthinking::dbeta2()` function. The nice thing about this parameterization is how intuitive the `pbar` parameter is. If you want a beta with an average of .2, you set `pbar <- .2`. If you want the distribution to be more or less certain, make the `theta` argument more or less large, respectively.

However, the beta density is typically defined in terms of α and β . If you denote the data as y , this follows the form

$$\text{Beta}(y|\alpha, \beta) = \frac{y^{\alpha-1}(1-y)^{\beta-1}}{\text{B}(\alpha, \beta)}$$

which you can verify in the *Continuous Distributions on [0, 1]* section of the [Stan Functions Reference](#). In the formula, B stands for the Beta function, which computes a normalizing constant, which you can learn about in the *Mathematical*

Functions of the Stan reference manual. This is all important to be aware of because when we defined that beta prior for `zi` in the last model, it was using this parameterization. Also, if you look at the base R `dbeta()` function, you'll learn it takes two parameters, `shape1` and `shape2`. Those uncreatively-named parameters are the same α and β from the density, above. They do not correspond to the `pbar` and `theta` parameters of McElreath's `rethinking::dbeta2()`.

McElreath had good reason for using `dbeta2()`. Beta's typical α and β parameters aren't the most intuitive to use; the parameters in McElreath's `dbeta2()` are much nicer. But if you're willing to dive deeper, it turns out you can find the mean of a beta distribution in terms of α and β like this

$$\mu = \frac{\alpha}{\alpha + \beta}$$

We can talk about the spread of the distribution, sometimes called κ , in terms α and β like this

$$\kappa = \alpha + \beta$$

With μ and κ in hand, we can even find the *SD* of a beta distribution with

$$\sigma = \sqrt{\mu(1 - \mu)/(\kappa + 1)}$$

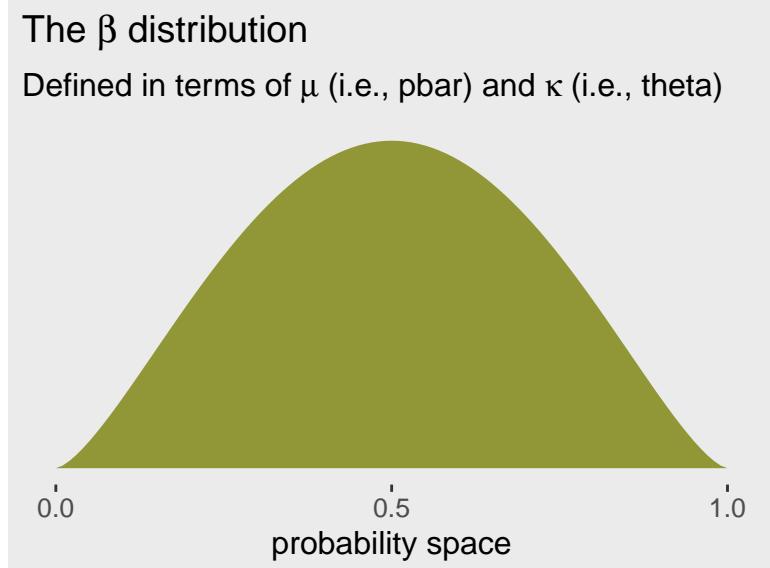
I explicate all this because McElreath's `pbar` is $\mu = \frac{\alpha}{\alpha + \beta}$ and his `theta` is $\kappa = \alpha + \beta$. This is great news because it means that we can understand what McElreath did with his `beta2()` function in terms of base R's `dbeta()` function. Which also means that we can understand the distribution of the beta parameters used in `brms::brm()`. To demonstrate, let's walk through McElreath's R code 11.25.

```
pbar <- 0.5
theta <- 5

ggplot(data = tibble(x = seq(from = 0, to = 1, by = .01))) +
  geom_ribbon(aes(x      = x,
                   ymin = 0,
                   ymax = rethinking::dbeta2(x, pbar, theta)),
              fill = canva_pal("Green fields")(4)[1]) +
  scale_x_continuous("probability space", breaks = c(0, .5, 1)) +
  scale_y_continuous(NULL, breaks = NULL) +
  ggtitle(expression(paste("The ", beta, " distribution"))),
  subtitle = expression(paste("Defined in terms of ", mu, " (i.e., pbar) and ", kappa, " (i.e., theta)")),
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))
```

The β distribution

Defined in terms of μ (i.e., `pbar`) and κ (i.e., `theta`)



In his 2014 text, *Doing Bayesian Data Analysis*, Kruschke provided code for a convenience function that will take `pbar` and `theta` as inputs and return the corresponding α and β values. Here's the function:

```
betaABfromMeanKappa <- function(mean, kappa) {
  if (mean <= 0 | mean >= 1) stop("must have 0 < mean < 1")
  if (kappa <= 0) stop("kappa must be > 0")
  a <- mean * kappa
  b <- (1.0 - mean) * kappa
  return(list(a = a, b = b))
}
```

Now we can use Kruschke's `betaABfromMeanKappa()` to find the α and β values corresponding to `pbar` and `theta`.

```
betaABfromMeanKappa(mean = pbar, kappa = theta)
```

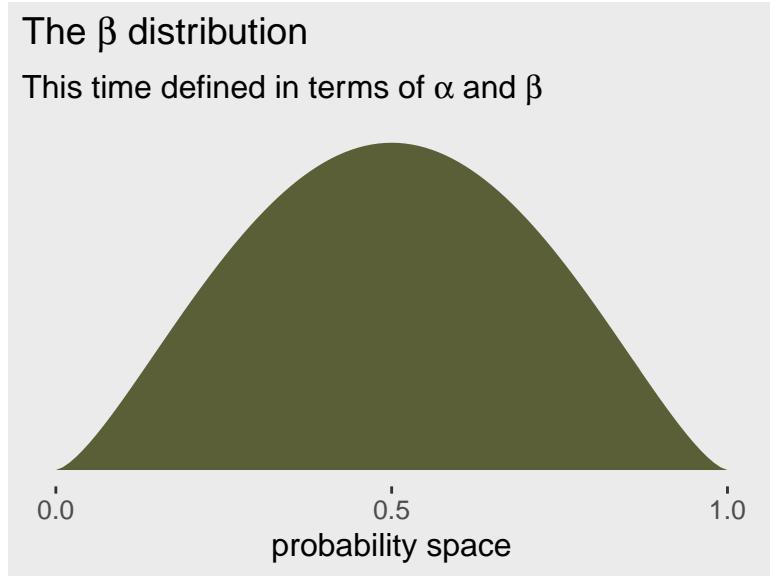
```
## $a
## [1] 2.5
##
## $b
## [1] 2.5
```

And finally, we can double check that all of this works. Here's the same distribution but defined in terms of α and β .

```
ggplot(data = tibble(x = seq(from = 0, to = 1, by = .01))) +
  geom_ribbon(aes(x      = x,
                  ymin   = 0,
                  ymax   = dbeta(x, 2.5, 2.5)),
              fill = canva_pal("Green fields")(4)[4]) +
  scale_x_continuous("probability space", breaks = c(0, .5, 1)) +
  scale_y_continuous(NULL, breaks = NULL) +
  ggtitle(expression(paste("The ", beta, " distribution"))),
  subtitle = expression(paste("This time defined in terms of ", alpha, " and ", beta))) +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))
```

The β distribution

This time defined in terms of α and β



McElreath encouraged us to “explore different values for `pbar` and `theta`” (p. 348). Here's a grid of plots with `pbar = c(.25, .5, .75)` and `theta = c(5, 10, 15)`

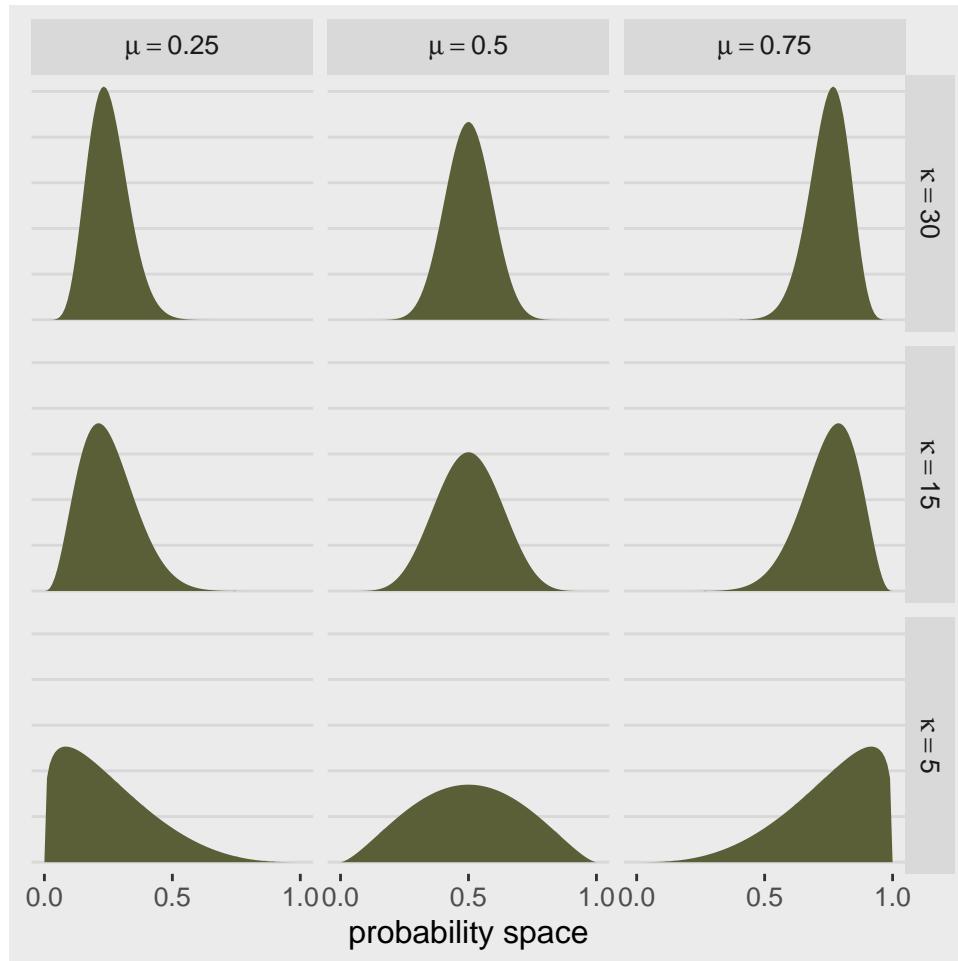
```
# data
tibble(pbar = c(.25, .5, .75)) %>%
  expand(pbar, theta = c(5, 10, 15)) %>%
  expand(nesting(pbar, theta), x = seq(from = 0, to = 1, length.out = 100)) %>%
```

```

mutate(density = rethinking::dbeta2(x, pbar, theta),
       mu      = str_c("mu == ", pbar %>% str_remove(., "0")),
       kappa   = str_c("kappa == ", theta)) %>%
mutate(kappa = factor(kappa, levels = c("kappa == 30", "kappa == 15", "kappa == 5"))) %>%

# plot
ggplot() +
  geom_ribbon(aes(x     = x,
                  ymin = 0,
                  ymax = density),
              fill = canva_pal("Green fields")(4)[4]) +
  scale_x_continuous("probability space", breaks = c(0, .5, 1)) +
  scale_y_continuous(NULL, labels = NULL) +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        axis.ticks.y = element_blank()) +
  facet_grid(kappa ~ mu, labeller = label_parsed)

```



If you'd like to see how to make a similar plot in terms of α and β , see [the chapter 6 document](#) of my project [recoding Kruschke's text into tidyverse and brms code](#).

But remember, we're not fitting a beta model. We're using the beta-binomial. “We’re going to bind our linear model to \bar{p} , so that changes in predictor variables change the central tendency of the distribution” (p. 348). The statistical model we’ll be fitting follows the form

$$\begin{aligned}\text{admit}_i &\sim \text{BetaBinomial}(n_i, \bar{p}_i, \theta) \\ \text{logit}(\bar{p}_i) &= \alpha \\ \alpha &\sim \text{Normal}(0, 2) \\ \theta &\sim \text{Exponential}(1)\end{aligned}$$

Here the size $n = \text{applications}$. In case you're confused, yes, our statistical model is not the one McElreath presented at the top of page 348 in the text. If you look closely, the statistical formula he presented does not match up with the one implied by his R code 11.26. Our statistical formula and the `brm()` model we'll be fitting, below, correspond to his R code 11.26.

Before we fit, we have an additional complication. The beta-binomial distribution is [not implemented in brms at this time](#). However, brms versions 2.2.0 and above allow users to define custom distributions. You can find the handy [vignette here](#). Happily, Bürkner even used the [beta-binomial distribution as the exemplar](#) in the vignette.

Before we get carried away, let's load the data.

```
library(rethinking)
data(UCBadmit)
d <- UCBadmit
```

Unload rethinking and load brms.

```
rm(UCBadmit)
detach(package:rethinking, unload = T)
library(brms)
```

I'm not going to go into great detail explaining the ins and outs of making custom distributions for `brm()`. You've got Bürkner's vignette for that. For our purposes, we need two preparatory steps. First, we need to use the `custom_family()` function to define the name and parameters of the beta-binomial distribution for use in `brm()`. Second, we have to define some functions for Stan which are not defined in Stan itself. We'll save them as `stan_funs`. Third, we'll make a `stanvar()` statement which will allow us to pass our `stan_funs` to `brm()`.

```
beta_binomial2 <-
  custom_family(
    "beta_binomial2", dpars = c("mu", "phi"),
    links = c("logit", "log"), lb = c(NA, 0),
    type = "int", vars = "trials[n]"
  )

stan_funs <- "
  real beta_binomial2_lpmf(int y, real mu, real phi, int T) {
    return beta_binomial_lpmf(y | T, mu * phi, (1 - mu) * phi);
  }
  int beta_binomial2_rng(real mu, real phi, int T) {
    return beta_binomial_rng(T, mu * phi, (1 - mu) * phi);
  }
"

stanvars <-
  stanvar(scode = stan_funs, block = "functions")
```

With that out of the way, we're almost ready to test this baby out. Before we do, a point of clarification: What McElreath referred to as the shape parameter, θ , Bürkner called the precision parameter, ϕ . In our exposition, above, we followed Kruschke's convention and called it κ . These are all the same thing: θ , ϕ , and κ are all the same thing. Perhaps less confusingly, what McElreath called the `pbar` parameter, \bar{p} , Bürkner simply called μ .

```
b11.5 <-
  brm(data = d,
    family = beta_binomial2, # here's our custom likelihood
    admit | trials(applications) ~ 1,
    prior = c(prior(normal(0, 2), class = Intercept),
              prior(exponential(1), class = phi)),
    iter = 4000, warmup = 1000, cores = 2, chains = 2,
    stanvars = stanvars, # note our `stanvars`
    seed = 11)
```

Success, our results look a lot like those in the text!

```
print(b11.5)
```

```
## Family: beta_binomial2
##   Links: mu = logit; phi = identity
## Formula: admit | trials(applications) ~ 1
##   Data: d (Number of observations: 12)
## Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##           total post-warmup samples = 6000
##
## Population-Level Effects:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept     -0.38      0.31    -0.99     0.24        4058  1.00
##
## Family Specific Parameters:
##               Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## phi          2.77      0.95     1.29     4.97        3663  1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Here's what the corresponding `posterior_samples()` data object looks like.

```
post <- posterior_samples(b11.5)

head(post)
```

```
##   b_Intercept     phi     lp_-
## 1 -0.48894704 3.660582 -70.37537
## 2  0.04534325 4.072111 -72.44047
## 3 -0.57268470 2.827876 -70.24965
## 4 -0.07413730 1.750735 -71.15664
## 5 -0.52303155 3.324634 -70.24396
## 6 -0.27625574 2.279246 -70.24819
```

Here's our median and percentile-based 95% interval.

```
post %>%
  tidybayes::median_qi(inv_logit_scaled(b_Intercept)) %>%
  mutate_if(is.double, round, digits = 3)

##   inv_logit_scaled(b_Intercept) .lower .upper .width .point .interval
## 1                           0.406  0.271  0.559   0.95 median      qi
```

To stay within the tidyverse while making the many thin lines in Figure 11.5.a, we're going to need to do a bit of data processing. First, we'll want a variable to index the rows in `post` (i.e., to index the posterior draws). And we'll want to convert the `b_Intercept` to the \bar{p} metric with the `inv_logit_scaled()` function. Then we'll use `sample_n()` to randomly draw a subset of the posterior draws. Then with the `expand()` function, we'll insert a tightly-spaced sequence of `x` values ranging between 0 and 1—the parameter space of beta distribution. Finally, we'll use `pmap_dbl()` to compute the density values for the `rethinking::dbeta2` distribution corresponding to the unique combination of `x`, `p_bar`, and `phi` values in each row.

```
set.seed(11)

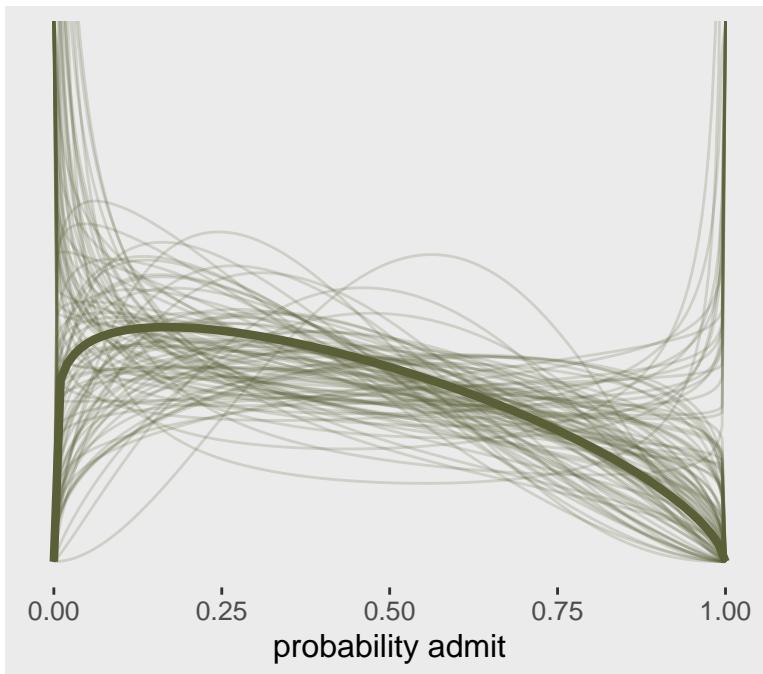
lines <-
  post %>%
  mutate(iter = 1:n(),
        p_bar = inv_logit_scaled(b_Intercept)) %>%
  sample_n(size = 100) %>%
  expand(nesting(iter, p_bar, phi),
        x = seq(from = 0, to = 1, by = .005)) %>%
  mutate(density = pmap_dbl(list(x, p_bar, phi), rethinking::dbeta2))

str(lines)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    20100 obs. of  5 variables:
## $ iter   : int  4 4 4 4 4 4 4 4 4 ...
## $ p_bar  : num  0.481 0.481 0.481 0.481 0.481 ...
## $ phi    : num  1.75 1.75 1.75 1.75 1.75 ...
## $ x      : num  0 0.005 0.01 0.015 0.02 0.025 0.03 0.035 0.04 0.045 ...
## $ density: num  Inf 1.78 1.59 1.5 1.43 ...
```

All that was just for the thin lines. To make the thicker line for the posterior mean, we'll get tricky with `stat_function()`.

```
lines %>%
  ggplot(aes(x = x, y = density)) +
  stat_function(fun = rethinking::dbeta2,
                args = list(prob = mean(inv_logit_scaled(post[, 1])),
                            theta = mean(post[, 2])),
                color = canva_pal("Green fields")(4)[4],
                size = 1.5) +
  geom_line(aes(group = iter),
            alpha = .2, color = canva_pal("Green fields")(4)[4]) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(ylim = 0:3) +
  xlab("probability admit") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))
```



There are other ways to do this. For ideas, check out my blog post [Make rotated Gaussians, Kruschke style](#).

Before we can do our variant of Figure 11.5.b, we'll need to define a few more custom functions. The `log_lik_beta_binomial2()` and `predict_beta_binomial2()` functions are required for `brms::predict()` to work with our `family = beta_binomial2` `brmfit` object. Similarly, `fitted_beta_binomial2()` is required for `brms::fitted()` to work properly. And before all that, we need to throw in a line with the `expose_functions()` function. Just go with it.

```
expose_functions(b11.5, vectorize = TRUE)

# required to use `predict()`
log_lik_beta_binomial2 <-
  function(i, draws) {
    mu <- draws$dpars$mu[, i]
    phi <- draws$dpars$phi
    N <- draws$data$trials[i]
    y <- draws$data$Y[i]
    beta_binomial2_lpmf(y, mu, phi, N)
  }

predict_beta_binomial2 <-
  function(i, draws, ...) {
    mu <- draws$dpars$mu[, i]
    phi <- draws$dpars$phi
    N <- draws$data$trials[i]
    beta_binomial2_rng(mu, phi, N)
  }

# required to use `fitted()`
fitted_beta_binomial2 <-
  function(draws) {
    mu      <- draws$dpars$mu
    trials <- draws$data$trials
    trials <- matrix(trials, nrow = nrow(mu), ncol = ncol(mu), byrow = TRUE)
    mu * trials
  }
```

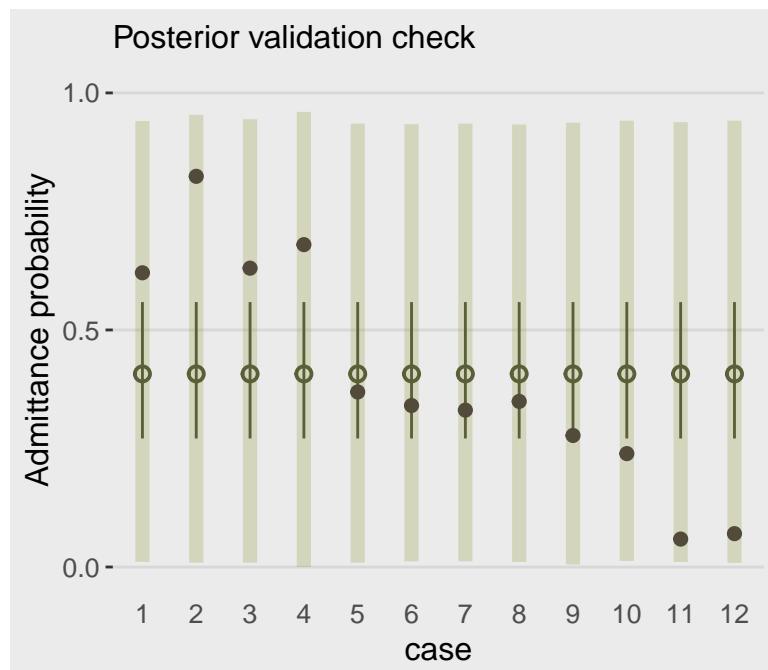
With those intermediary steps out of the way, we're ready to make Figure 11.5.b.

```

# the prediction intervals
predict(b11.5) %>%
  as_tibble() %>%
  transmute(ll = Q2.5,
            ul = Q97.5) %>%
# the fitted intervals
bind_cols(
  fitted(b11.5) %>%
  as_tibble()
) %>%
# The original data used to fit the model
bind_cols(b11.5$data) %>%
mutate(case = 1:12) %>%

# plot
ggplot(aes(x = case)) +
  geom_linerange(aes(ymin = ll / applications,
                      ymax = ul / applications),
                 color = canva_pal("Green fields")(4)[1],
                 size = 2.5, alpha = 1/4) +
  geom_pointrange(aes(ymin = Q2.5 / applications,
                       ymax = Q97.5 / applications,
                       y = Estimate/applications),
                  color = canva_pal("Green fields")(4)[4],
                  size = 1/2, shape = 1) +
  geom_point(aes(y = admit/applications),
             color = canva_pal("Green fields")(4)[2],
             size = 2) +
  scale_x_continuous(breaks = 1:12) +
  scale_y_continuous(breaks = c(0, .5, 1)) +
  coord_cartesian(ylim = 0:1) +
  labs(subtitle = "Posterior validation check",
       y      = "Admittance probability") +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"),
        axis.ticks.x = element_blank(),
        legend.position = "none")

```



As in the text, the raw data are consistent with the prediction intervals. But those intervals are so incredibly wide, they're hardly an endorsement of the model. Once we learn about hierarchical models, we'll be able to do much better.

11.3.2 Negative-binomial or gamma-Poisson.

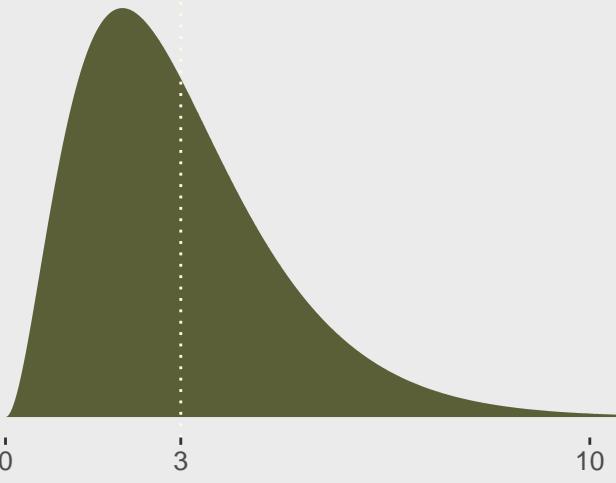
Recall the Poisson distribution presumes σ^2 scales with μ . The negative binomial distribution relaxes this assumption and presumes “each Poisson count observation has its own rate. It estimates the shape of a gamma distribution to describe the Poisson rates across cases” (p. 350).

Here's a look at the γ distribution.

```
mu     <- 3
theta <- 1

ggplot(data = tibble(x = seq(from = 0, to = 12, by = .01)),
       aes(x = x)) +
  geom_ribbon(aes(ymin = 0,
                   ymax = rethinking::dgamma2(x, mu, theta)),
              color = "transparent",
              fill = canva_pal("Green fields")(4)[4]) +
  geom_vline(xintercept = mu, linetype = 3,
             color = canva_pal("Green fields")(4)[3]) +
  scale_x_continuous(NULL, breaks = c(0, mu, 10)) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:10) +
  ggtitle(expression(paste("Our sweet ", gamma, "(3, 1)"))) +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))
```

Our sweet $\gamma(3, 1)$



11.3.2.1 Bonus: Let's fit a negative-binomial model.

McElreath didn't give an example of negative-binomial regression in the text. Here's one with the UCBadmit data.

```
brm(data = d, family = negbinomial,
     admit ~ 1 + applicant.gender,
     prior = c(prior(normal(0, 10), class = Intercept),
               prior(normal(0, 1), class = b),
               prior(gamma(0.01, 0.01), class = shape)), # this is the brms default
     iter = 4000, warmup = 1000, cores = 2, chains = 2,
```

```

seed = 11) %>%
print()

## Family: negbinomial
## Links: mu = log; shape = identity
## Formula: admit ~ 1 + applicant.gender
## Data: d (Number of observations: 12)
## Samples: 2 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 6000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      4.70     0.40    3.99    5.58      4439 1.00
## applicant.gendermale  0.57     0.50   -0.43    1.52      4929 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## shape       1.23     0.48    0.50    2.36      4408 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Since the negative-binomial model uses the log link, you need to exponentiate to get the estimates back into the count metric. E.g.,

```
exp(4.7)
```

```
## [1] 109.9472
```

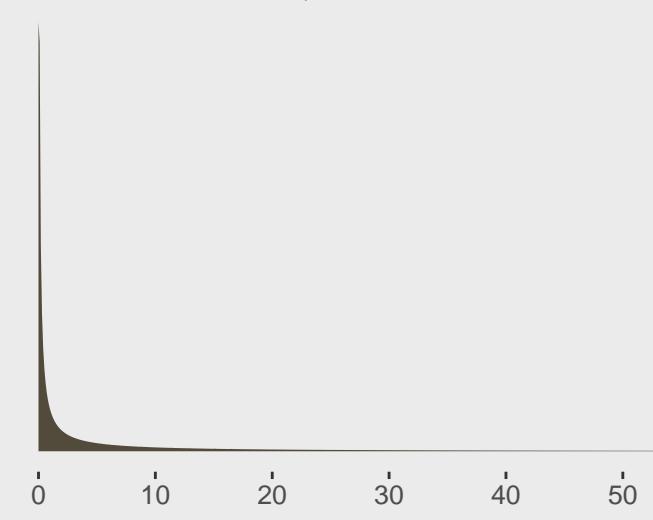
Also, you may have noticed we used the brms default `prior(gamma(0.01, 0.01), class = shape)` for the shape parameter. Here's what that prior looks like.

```

ggplot(data = tibble(x = seq(from = 0, to = 60, by = .1)),
       aes(x = x)) +
  geom_ribbon(aes(ymin = 0,
                  ymax = dgamma(x, 0.01, 0.01)),
              color = "transparent",
              fill = canva_pal("Green fields")(4)[2]) +
  scale_x_continuous(NULL) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = 0:50) +
  ggtitle(expression(paste("Our brms default ", gamma, "(0.01, 0.01) prior"))) +
  theme_hc() +
  theme(plot.background = element_rect(fill = "grey92"))

```

Our brms default $\gamma(0.01, 0.01)$ prior



11.3.3 Over-dispersion, entropy, and information criteria.

Both the beta-binomial and the gamma-Poisson models are maximum entropy for the same constraints as the regular binomial and Poisson. They just try to account for unobserved heterogeneity in probabilities and rates. So while they can be a lot harder to fit to data, they can be usefully conceptualized much like ordinary binomial and Poisson GLMs. So in terms of model comparison using information criteria, a beta-binomial model is a binomial model, and a gamma-Poisson (negative-binomial) is a Poisson model. (pp. 350–351)

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets  methods
## [8] base
##
## other attached packages:
## [1] gridExtra_2.3        broom_0.5.1          ggthemes_4.0.1
## [2]forcats_0.3.0        stringr_1.4.0        dplyr_0.8.0.1
## [3]purrr_0.2.5          readr_1.1.1          tidyrr_0.8.1
```

```
## [10] tibble_2.1.1           tidyverse_1.2.1       brms_2.8.8
## [13] Rcpp_0.1.0              rstan_2.18.2         StanHeaders_2.18.0-1
## [16] ggplot2_3.1.1
##
## loaded via a namespace (and not attached):
## [1] nlme_3.1-137            matrixStats_0.54.0
## [3] xts_0.10-2              lubridate_1.7.4
## [5] threejs_0.3.1           httr_1.3.1
## [7] rprojroot_1.3-2         tools_3.5.1
## [9] backports_1.1.4          utf8_1.1.4
## [11] R6_2.3.0                DT_0.4
## [13] lazyeval_0.2.2          colorspace_1.3-2
## [15] withr_2.1.2             tidyselect_0.2.5
## [17] prettyunits_1.0.2        processx_3.2.1
## [19] Brobdingnag_1.2-6       compiler_3.5.1
## [21] rvest_0.3.2              cli_1.0.1
## [23] arrayhelpers_1.0-20160527 xml2_1.2.0
## [25] shinyjs_1.0              labeling_0.3
## [27] colourpicker_1.0         bookdown_0.9
## [29] scales_1.0.0             dygraphs_1.1.1.5
## [31] mvtnorm_1.0-10           ggridges_0.5.0
## [33] callr_3.1.0              digest_0.6.18
## [35] rmarkdown_1.10            rethinking_1.80
## [37] base64enc_0.1-3          pkgconfig_2.0.2
## [39] htmltools_0.3.6          readxl_1.1.0
## [41] htmlwidgets_1.2           rlang_0.3.4
## [43] rstudioapi_0.7            shiny_1.1.0
## [45] svUnit_0.7-12            generics_0.0.2
## [47] zoo_1.8-2                jsonlite_1.5
## [49] crosstalk_1.0.0          gtools_3.8.1
## [51] inline_0.3.15            magrittr_1.5
## [53] loo_2.1.0                bayesplot_1.6.0
## [55] Matrix_1.2-14            fansi_0.4.0
## [57] munsell_0.5.0            abind_1.4-5
## [59] stringi_1.4.3            yaml_2.1.19
## [61] MASS_7.3-50               ggstance_0.3
## [63] pkgbuild_1.0.2            plyr_1.8.4
## [65] grid_3.5.1                promises_1.0.1
## [67] crayon_1.3.4              miniUI_0.1.1.1
## [69] lattice_0.20-35           haven_1.1.2
## [71] hms_0.4.2                knitr_1.20
## [73] ps_1.2.1                 pillar_1.3.1
## [75] igraph_1.2.1              markdown_0.8
## [77] shinystan_2.5.0            reshape2_1.4.3
## [79] stats4_3.5.1              rstantools_1.5.1
## [81] glue_1.3.1.9000            evaluate_0.10.1
## [83] modelr_0.1.2              httpuv_1.4.4.2
## [85] cellranger_1.1.0           gtable_0.3.0
## [87] assertthat_0.2.0            xfun_0.3
## [89] mime_0.5                  xtable_1.8-2
## [91] coda_0.19-2                later_0.7.3
## [93] rsconnect_0.8.8            shinythemes_1.1.1
## [95] tidybayes_1.0.4            bridgesampling_0.6-0
```


Chapter 12

Multilevel Models

Multilevel models... remember features of each cluster in the data as they learn about all of the clusters. Depending upon the variation among clusters, which is learned from the data as well, the model pools information across clusters. This pooling tends to improve estimates about each cluster. This improved estimation leads to several, more pragmatic sounding, benefits of the multilevel approach. (p. 356)

These benefits include:

- improved estimates for repeated sampling (i.e., in longitudinal data)
- improved estimates when there are imbalances among subsamples
- estimates of the variation across subsamples
- avoiding simplistic averaging by retaining variation across subsamples

All of these benefits flow out of the same strategy and model structure. You learn one basic design and you get all of this for free.

When it comes to regression, multilevel regression deserves to be the default approach. There are certainly contexts in which it would be better to use an old-fashioned single-level model. But the contexts in which multilevel models are superior are much more numerous. It is better to begin to build a multilevel analysis, and then realize it's unnecessary, than to overlook it. And once you grasp the basic multilevel stragety, it becomes much easier to incorporate related tricks such as allowing for measurement error in the data and even model missing data itself (Chapter 14). (p. 356)

I'm totally on board with this. After learning about the multilevel model, I see it everywhere. For more on the sentiment it should be the default, check out McElreath's blog post, [Multilevel Regression as Default](#).

12.1 Example: Multilevel tadpoles

Let's get the `reedfrogs` data from rethinking.

```
library(rethinking)

## Warning: package 'ggplot2' was built under R version 3.5.2

data(reedfrogs)
d <- reedfrogs
```

Detach rethinking and load brms.

```
rm(reedfrogs)
detach(package:rethinking, unload = T)
library(brms)
```

```
## Warning: package 'Rcpp' was built under R version 3.5.2
```

Go ahead and acquaint yourself with the `reedfrogs`.

```
library(tidyverse)
```

```
d %>%
```

```
glimpse()
```

```
## Observations: 48
```

```
## Variables: 5
```

```
## $ density <int> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 25, 25, 25, 25...
## $ pred <fct> no, no, no, no, no, no, no, pred, pred...
## $ size <fct> big, big, big, big, small, small, small, small, big, big, big, big, small, small, small, sma...
## $ surv <int> 9, 10, 7, 10, 9, 9, 10, 9, 4, 9, 7, 6, 7, 5, 9, 9, 24, 23, 22, 25, 23, 23, 23, ...
## $ propsurv <dbl> 0.9000000, 1.0000000, 0.7000000, 1.0000000, 0.9000000, 0.9000000, 1.0000000, 0...
```

Making the `tank` cluster variable is easy.

```
d <-  
d %>%  
mutate(tank = 1:nrow(d))
```

Here's the formula for the un-pooled model in which each `tank` gets its own intercept.

$$\begin{aligned} \text{surv}_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{tank}_i} \\ \alpha_{\text{tank}} &\sim \text{Normal}(0, 5) \end{aligned}$$

And $n_i = \text{density}_i$. Now we'll fit this simple aggregated binomial model much like we practiced in Chapter 10.

```
b12.1 <-  
brm(data = d, family = binomial,  
    surv | trials(density) ~ 0 + factor(tank),  
    prior(normal(0, 5), class = b),  
    iter = 2000, warmup = 500, chains = 4, cores = 4,  
    seed = 12)
```

The formula for the multilevel alternative is

$$\begin{aligned} \text{surv}_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{tank}_i} \\ \alpha_{\text{tank}} &\sim \text{Normal}(\alpha, \sigma) \\ \alpha &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

You specify the corresponding multilevel model like this.

```
b12.2 <-  
brm(data = d, family = binomial,  
    surv | trials(density) ~ 1 + (1 | tank),  
    prior = c(prior(normal(0, 1), class = Intercept),  
              prior(cauchy(0, 1), class = sd)),  
    iter = 4000, warmup = 1000, chains = 4, cores = 4,  
    seed = 12)
```

The syntax for the varying effects follows the [lme4 style](#), (`<varying parameter(s)> | <grouping variable(s)>`). In this case `(1 | tank)` indicates only the intercept, 1, varies by `tank`. The extent to which parameters vary is controlled by the prior, `prior(cauchy(0, 1), class = "sd")`, which is parameterized in the standard deviation metric. Do note that last part. It's common in multilevel software to model in the variance metric, instead.

Let's do the WAIC comparisons.

```
b12.1 <- add_criterion(b12.1, "waic")
b12.2 <- add_criterion(b12.2, "waic")

w <- loo_compare(b12.1, b12.2, criterion = "waic")

print(w, simplify = F)

##      elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic    se_waic
## b12.2     0.0      0.0 -100.0       3.6      20.9      0.8    200.0     7.2
## b12.1   -1.1      2.3 -101.0       4.7      22.9      0.7    202.1     9.4
```

The `se_diff` is large relative to the `elpd_diff`. If we convert the elpd difference to the WAIC metric, the message stays the same.

```
cbind(waic_diff = w[, 1] * -2,
      se        = w[, 2] * 2)

##      waic_diff      se
## b12.2  0.000000 0.000000
## b12.1  2.132735 4.555047
```

I'm not going to show it here, but if you'd like a challenge, try comparing the models with the LOO. You'll learn all about high `pareto_k` values, `kfold()` recommendations, and challenges implementing those `kfold()` recommendations. If you're interested, pour yourself a calming adult beverage, execute the code below, and check out the [“Kfold\(\): “Error: New factor levels are not allowed”](#) thread in the Stan forums.

```
b12.1 <- add_criterion(b12.1, "loo")
b12.2 <- add_criterion(b12.2, "loo")
```

But back on track, here's our prep work for Figure 12.1.

```
post <- posterior_samples(b12.2, add_chain = T)

post_mdn <-
  coef(b12.2, robust = T)$tank[, , ] %>%
  as_tibble() %>%
  bind_cols(d) %>%
  mutate(post_mdn = inv_logit_scaled(Estimate))

post_mdn

## # A tibble: 48 x 11
##      Estimate Est.Error   Q2.5 Q97.5 density pred  size    surv propsurv tank post_mdn
##      <dbl>     <dbl>  <dbl> <dbl>  <int> <fct> <fct> <int>    <dbl> <int>    <dbl>
## 1     2.08     0.850  0.589  4.09     10 no    big     9     0.9     1     0.889
## 2     2.95     1.05   1.19   5.44     10 no    big    10     1     2     0.950
## 3     0.970    0.652  -0.255  2.39     10 no    big     7     0.7     3     0.725
## 4     2.95     1.06   1.19   5.52     10 no    big    10     1     4     0.950
## 5     2.07     0.856  0.567  4.02     10 no   small    9     0.9     5     0.888
## 6     2.06     0.832  0.602  4.00     10 no   small    9     0.9     6     0.887
## 7     2.96     1.09   1.17   5.51     10 no   small   10     1     7     0.951
```

```

## 8    2.05      0.842  0.598 3.96      10 no     small     9      0.9      8      0.886
## 9   -0.183     0.606 -1.41   0.992      10 pred    big      4      0.4      9      0.454
## 10   2.06      0.839  0.596 4.05      10 pred    big      9      0.9      10     0.887
## # ... with 38 more rows

```

For kicks and giggles, let's use a [FiveThirtyEight-like theme](#) for this chapter's plots. An easy way to do so is with help from the [ggthemes package](#).

```

# install.packages("ggthemes", dependencies = T)
library(ggthemes)

```

Finally, here's the ggplot2 code to reproduce Figure 12.1.

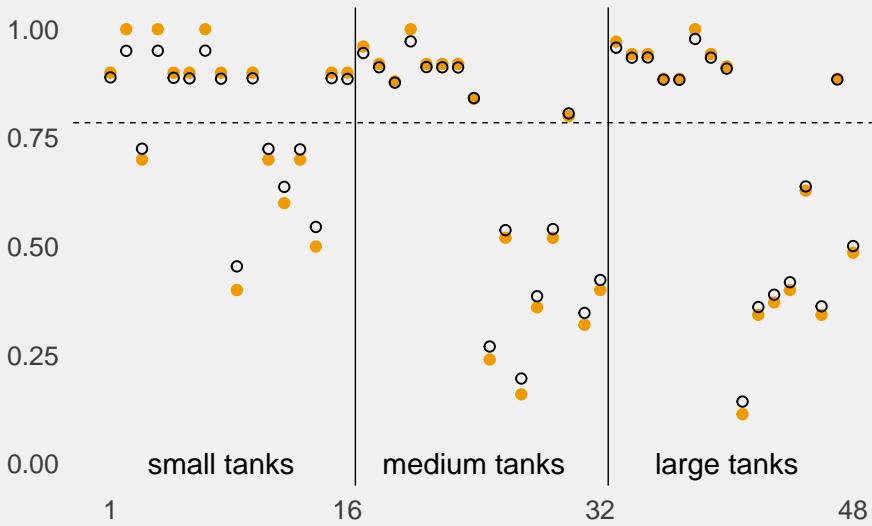
```

post_mdn %>%
  ggplot(aes(x = tank)) +
  geom_hline(yintercept = inv_logit_scaled(median(post$b_Intercept)), linetype = 2, size = 1/4) +
  geom_vline(xintercept = c(16.5, 32.5), size = 1/4) +
  geom_point(aes(y = propsurv), color = "orange2") +
  geom_point(aes(y = post_mdn), shape = 1) +
  coord_cartesian(ylim = c(0, 1)) +
  scale_x_continuous(breaks = c(1, 16, 32, 48)) +
  labs(title = "Multilevel shrinkage!",
       subtitle = "The empirical proportions are in orange while the model-implied proportions are the black circles. The dashed line is the model-implied average survival proportion.",
       annotate("text", x = c(8, 16 + 8, 32 + 8), y = 0,
               label = c("small tanks", "medium tanks", "large tanks")) +
  theme_fivethirtyeight() +
  theme(panel.grid = element_blank())

```

Multilevel shrinkage!

The empirical proportions are in orange while the model-implied proportions are the black circles. The dashed line is the model-implied average survival proportion.



Here is our version of Figure 12.2.a.

```

# this makes the output of `sample_n()` reproducible
set.seed(12)

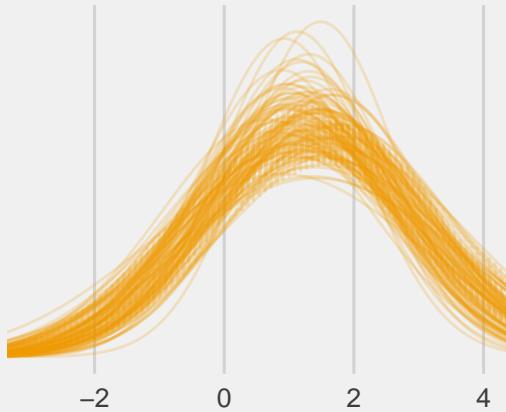
```

```
post %>%
  sample_n(100) %>%
  expand(nesting(iter, b_Intercept, sd_tank__Intercept),
         x = seq(from = -4, to = 5, length.out = 100)) %>%

  ggplot(aes(x = x, group = iter)) +
  geom_line(aes(y = dnorm(x, b_Intercept, sd_tank__Intercept)),
            alpha = .2, color = "orange2") +
  labs(title = "Population survival distribution",
       subtitle = "The Gaussians are on the log-odds scale.") +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = c(-3, 4)) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 13),
        plot.subtitle = element_text(size = 10))
```

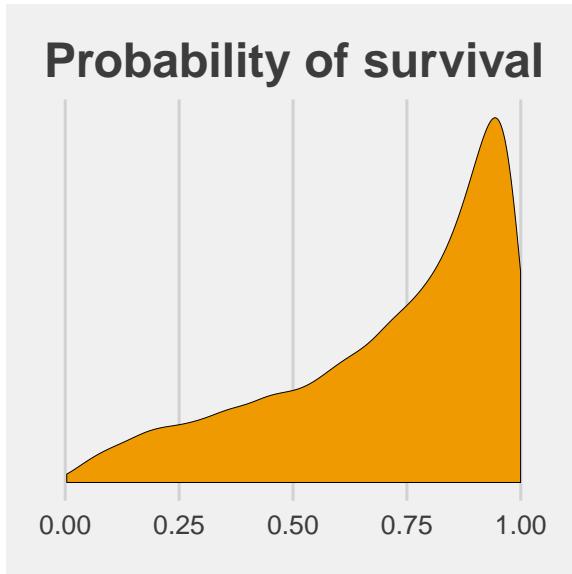
Population survival distribution

The Gaussians are on the log-odds scale.



Note the uncertainty in terms of both location α and scale σ . Now here's the code for Figure 12.2.b.

```
ggplot(data = post,
       aes(x = rnorm(n      = nrow(post),
                     mean = b_Intercept,
                     sd   = sd_tank__Intercept) %>%
             inv_logit_scaled()) +
       geom_density(size = 0, fill = "orange2") +
       scale_y_continuous(NULL, breaks = NULL) +
       ggtitle("Probability of survival") +
       theme_fivethirtyeight()
```



Note how we sampled 12,000 imaginary `tanks` rather than McElreath's 8,000. This is because we had 12,000 HMC iterations (i.e., execute `nrow(post)`).

The `aes()` code, above, was a bit much. To get a sense of how it worked, consider this:

```
set.seed(12)

rnorm(n      = 1,
      mean = post$b_Intercept,
      sd   = post$sd_tank__Intercept) %>%
  inv_logit_scaled()

## [1] 0.2135091
```

First, we took one random draw from a normal distribution with a mean of the first row in `post$b_Intercept` and a standard deviation of the value from the first row in `post$sd_tank__Intercept`, and passed it through the `inv_logit_scaled()` function. By replacing the 1 with `nrow(post)`, we do this `nrow(post)` times (i.e., 12,000). Our orange density, then, is the summary of that process.

12.1.0.1 Overthinking: Prior for variance components.

Yep, you can use the exponential distribution for your priors in brms. Here it is for model `b12.2`.

```
b12.2.e <-
  update(b12.2,
         prior = c(prior(normal(0, 1), class = Intercept),
                    prior(exponential(1), class = sd)))
```

The model summary:

```
print(b12.2.e)

##  Family: binomial
##  Links: mu = logit
## Formula: surv | trials(density) ~ 1 + (1 | tank)
##  Data: d (Number of observations: 48)
##  Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 1;
##            total post-warmup samples = 12000
##
```

```

## Group-Level Effects:
## ~tank (Number of levels: 48)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)    1.61      0.22     1.24     2.08       2962 1.00
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      1.30      0.25     0.82     1.78       2304 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

If you're curious how the exponential prior compares to the posterior, you might just plot.

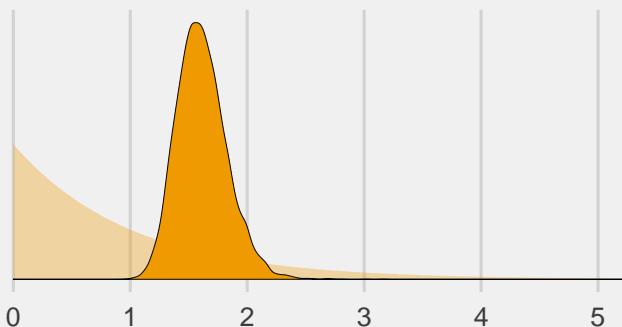
```

tibble(x = seq(from = 0, to = 6, by = .01)) %>%
  ggplot() +
  # the prior
  geom_ribbon(aes(x = x, ymin = 0, ymax = dexp(x, rate = 1)),
              fill = "orange2", alpha = 1/3) +
  # the posterior
  geom_density(data = posterior_samples(b12.2.e),
               aes(x = sd_tank_Intercept),
               fill = "orange2", size = 0) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = c(0, 5)) +
  labs(title = "Bonus prior/posterior plot\nfor sd_tank_Intercept",
       subtitle = "The prior is the semitransparent ramp in the\nbackground. The posterior is the solid orange")
  theme_fivethirtyeight()

```

Bonus prior/posterior plot for sd_tank_Intercept

The prior is the semitransparent ramp in the background. The posterior is the solid orange mound.



12.2 Varying effects and the underfitting/overfitting trade-off

Varying intercepts are just regularized estimates, but adaptively regularized by estimating how diverse the clusters are while estimating the features of each cluster. This fact is not easy to grasp...

A major benefit of using varying effects estimates, instead of the empirical raw estimates, is that they provide more accurate estimates of the individual cluster (tank) intercepts. On average, the varying effects actually provide a

better estimate of the individual tank (cluster) means. The reason that the varying intercepts provides better estimates is that they do a better job trading off underfitting and overfitting. (p. 364)

In this section, we explicate this by contrasting three perspectives:

- Complete pooling (i.e., a single- α model)
- No pooling (i.e., the single-level α_{tank_i} model)
- Partial pooling (i.e., the multilevel model for which $\alpha_{\text{tank}} \sim \text{Normal}(\alpha, \sigma)$)

To demonstrate [the magic of the multilevel model], we'll simulate some tadpole data. That way, we'll know the true per-pond survival probabilities. Then we can compare the no-pooling estimates to the partial pooling estimates, by computing how close each gets to the true values they are trying to estimate. The rest of this section shows how to do such a simulation. (p. 365)

12.2.1 The model.

The simulation formula should look familiar.

$$\begin{aligned}\text{surv}_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{pond}_i} \\ \alpha_{\text{pond}} &\sim \text{Normal}(\alpha, \sigma) \\ \alpha &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{HalfCauchy}(0, 1)\end{aligned}$$

12.2.2 Assign values to the parameters.

```
a      <- 1.4
sigma  <- 1.5
n_ponds <- 60

set.seed(12)
(
  dsim <-
  tibble(pond    = 1:n_ponds,
        ni      = rep(c(5, 10, 25, 35), each = n_ponds / 4) %>% as.integer(),
        true_a = rnorm(n = n_ponds, mean = a, sd = sigma))
)

## # A tibble: 60 x 3
##   pond    ni   true_a
##   <int> <int>   <dbl>
## 1     1     5 -0.821
## 2     2     5  3.77
## 3     3     5 -0.0351
## 4     4     5  0.0200
## 5     5     5 -1.60
## 6     6     5  0.992
## 7     7     5  0.927
## 8     8     5  0.458
## 9     9     5  1.24
## 10    10    5  2.04
## # ... with 50 more rows
```

12.2.3 Sumulate survivors.

Each pond i has n_i potential survivors, and nature flips each tadpole's coin, so to speak, with probability of survival p_i . This probability p_i is implied by the model definition, and is equal to:

$$p_i = \frac{\exp(\alpha_i)}{1 + \exp(\alpha_i)}$$

The model uses a logit link, and so the probability is defined by the [`inv_logit_scaled()`] function. (p. 367)

```
set.seed(12)
(
  dsim <-
  dsim %>%
  mutate(si = rbinom(n = n(), prob = inv_logit_scaled(true_a), size = ni))
)

## # A tibble: 60 x 4
##   pond    ni  true_a     si
##   <int> <int>   <dbl> <int>
## 1     1     5 -0.821     0
## 2     2     5  3.77      5
## 3     3     5 -0.0351    4
## 4     4     5  0.0200    3
## 5     5     5 -1.60      0
## 6     6     5  0.992      5
## 7     7     5  0.927      5
## 8     8     5  0.458      3
## 9     9     5  1.24       5
## 10   10     5  2.04       5
## # ... with 50 more rows
```

12.2.4 Compute the no-pooling estimates.

The no-pooling estimates (i.e., α_{tank_i}) are the results of simple algebra.

```
(

  dsim <-
  dsim %>%
  mutate(p_nopool = si / ni)
)

## # A tibble: 60 x 5
##   pond    ni  true_a     si p_nopool
##   <int> <int>   <dbl> <int>     <dbl>
## 1     1     5 -0.821     0      0
## 2     2     5  3.77      5      1
## 3     3     5 -0.0351    4      0.8
## 4     4     5  0.0200    3      0.6
## 5     5     5 -1.60      0      0
## 6     6     5  0.992      5      1
## 7     7     5  0.927      5      1
## 8     8     5  0.458      3      0.6
## 9     9     5  1.24       5      1
## 10   10     5  2.04       5      1
## # ... with 50 more rows
```

“These are the same no-pooling estimates you’d get by fitting a model with a dummy variable for each pond and flat priors that induce no regularization” (p. 367).

12.2.5 Compute the partial-pooling estimates.

To follow along with McElreath, set `chains = 1, cores = 1` to fit with one chain.

```
b12.3 <-  
  brm(data = dsim, family = binomial,  
    si | trials(ni) ~ 1 + (1 | pond),  
    prior = c(prior(normal(0, 1), class = Intercept),  
              prior(cauchy(0, 1), class = sd)),  
    iter = 10000, warmup = 1000, chains = 1, cores = 1,  
    seed = 12)  
  
print(b12.3)  
  
## Family: binomial  
## Links: mu = logit  
## Formula: si | trials(ni) ~ 1 + (1 | pond)  
## Data: dsim (Number of observations: 60)  
## Samples: 1 chains, each with iter = 10000; warmup = 1000; thin = 1;  
##           total post-warmup samples = 9000  
##  
## Group-Level Effects:  
## ~pond (Number of levels: 60)  
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## sd(Intercept)     1.30      0.19     0.97     1.71       2948 1.00  
##  
## Population-Level Effects:  
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## Intercept       1.28      0.20     0.90     1.67       2996 1.00  
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

I'm not aware that you can use McElreath's `depth=2` trick in `brms` for `summary()` or `print()`. But can get that information with the `coef()` function.

```
coef(b12.3)$pond[c(1:2, 59:60), , ] %>%  
  round(digits = 2)
```

```
##   Estimate Est.Error Q2.5 Q97.5  
## 1     -1.07      0.89 -3.03  0.54  
## 2      2.30      1.02  0.51  4.54  
## 59     0.97      0.37  0.27  1.72  
## 60     1.42      0.41  0.65  2.29
```

Note how we just peeked at the top and bottom two rows with the `c(1:2, 59:60)` part of the code, there. Somewhat discouragingly, `coef()` doesn't return the 'Eff.Sample' or 'Rhat' columns as in McElreath's output. We can still extract that information, though. For \hat{R} , the solution is simple; use the `brms::rhat()` function.

```
rhat(b12.3)
```

```
##          b_Intercept  sd_pond_Intercept  r_pond[1,Intercept]  r_pond[2,Intercept]  
##          1.0002418      0.9999371          0.9999115      1.0002275  
##  r_pond[3,Intercept]  r_pond[4,Intercept]  r_pond[5,Intercept]  r_pond[6,Intercept]  
##          0.9998903      0.9999139          0.9999526      0.9998954  
##  r_pond[7,Intercept]  r_pond[8,Intercept]  r_pond[9,Intercept]  r_pond[10,Intercept]
```

```

##          1.0002992      1.0000516      0.9999460      0.9999920
## r_pond[11,Intercept] r_pond[12,Intercept] r_pond[13,Intercept] r_pond[14,Intercept]
##          0.9999889      0.9999319      0.9999591      0.9999101
## r_pond[15,Intercept] r_pond[16,Intercept] r_pond[17,Intercept] r_pond[18,Intercept]
##          1.0001250      0.9999149      0.9999016      0.9999239
## r_pond[19,Intercept] r_pond[20,Intercept] r_pond[21,Intercept] r_pond[22,Intercept]
##          0.9998892      0.9998931      0.9998939      0.9998921
## r_pond[23,Intercept] r_pond[24,Intercept] r_pond[25,Intercept] r_pond[26,Intercept]
##          0.9998933      0.9998986      0.9999875      0.9999691
## r_pond[27,Intercept] r_pond[28,Intercept] r_pond[29,Intercept] r_pond[30,Intercept]
##          0.9999052      0.9998893      0.9999064      1.0000005
## r_pond[31,Intercept] r_pond[32,Intercept] r_pond[33,Intercept] r_pond[34,Intercept]
##          1.0000856      0.9998893      0.9998950      0.9999871
## r_pond[35,Intercept] r_pond[36,Intercept] r_pond[37,Intercept] r_pond[38,Intercept]
##          0.9999385      1.0000467      0.9998907      0.9999261
## r_pond[39,Intercept] r_pond[40,Intercept] r_pond[41,Intercept] r_pond[42,Intercept]
##          0.9999002      0.9999136      0.9999024      0.9998960
## r_pond[43,Intercept] r_pond[44,Intercept] r_pond[45,Intercept] r_pond[46,Intercept]
##          0.9999817      1.0000738      1.0000156      1.0000022
## r_pond[47,Intercept] r_pond[48,Intercept] r_pond[49,Intercept] r_pond[50,Intercept]
##          0.9999479      0.9999159      0.9999162      0.9998972
## r_pond[51,Intercept] r_pond[52,Intercept] r_pond[53,Intercept] r_pond[54,Intercept]
##          0.9998896      0.9999483      0.9999232      0.9999950
## r_pond[55,Intercept] r_pond[56,Intercept] r_pond[57,Intercept] r_pond[58,Intercept]
##          0.9999233      0.9998901      0.9999382      0.9999012
## r_pond[59,Intercept] r_pond[60,Intercept]
##          0.9999175      0.9998905      0.9999222
##          lp__
##          0.9999175

```

Extracting the ‘Eff.Sample’ values is a little more complicated. There is no `effsamples()` function. However, we do have `neff_ratio()`.

`neff_ratio(b12.3)`

```

##          b_Intercept    sd_pond__Intercept    r_pond[1,Intercept]    r_pond[2,Intercept]
##          0.3328983      0.3276092      1.3086470      1.4222957
## r_pond[3,Intercept] r_pond[4,Intercept] r_pond[5,Intercept] r_pond[6,Intercept]
##          1.7287578      1.7931600      1.5172159      1.7087264
## r_pond[7,Intercept] r_pond[8,Intercept] r_pond[9,Intercept] r_pond[10,Intercept]
##          1.4415308      1.6367495      1.5541417      1.5271929
## r_pond[11,Intercept] r_pond[12,Intercept] r_pond[13,Intercept] r_pond[14,Intercept]
##          1.5474498      1.8682911      1.7837498      1.8735102
## r_pond[15,Intercept] r_pond[16,Intercept] r_pond[17,Intercept] r_pond[18,Intercept]
##          1.6606211      1.4500515      1.3307980      1.4573910
## r_pond[19,Intercept] r_pond[20,Intercept] r_pond[21,Intercept] r_pond[22,Intercept]
##          1.4950252      1.4288447      1.4943737      1.3833042
## r_pond[23,Intercept] r_pond[24,Intercept] r_pond[25,Intercept] r_pond[26,Intercept]
##          1.3976878      1.3691378      1.3661559      1.6933039
## r_pond[27,Intercept] r_pond[28,Intercept] r_pond[29,Intercept] r_pond[30,Intercept]
##          1.3845310      1.7058099      1.7976571      1.6092142
## r_pond[31,Intercept] r_pond[32,Intercept] r_pond[33,Intercept] r_pond[34,Intercept]
##          1.2267648      1.2972673      1.0352256      0.9503992
## r_pond[35,Intercept] r_pond[36,Intercept] r_pond[37,Intercept] r_pond[38,Intercept]
##          0.9909477      0.9580389      1.1905664      1.0429481
## r_pond[39,Intercept] r_pond[40,Intercept] r_pond[41,Intercept] r_pond[42,Intercept]
##          1.3957889      1.1026741      1.1539322      0.9819976
## r_pond[43,Intercept] r_pond[44,Intercept] r_pond[45,Intercept] r_pond[46,Intercept]
##          1.0965749      1.1089698      0.9528173      1.1863659
## r_pond[47,Intercept] r_pond[48,Intercept] r_pond[49,Intercept] r_pond[50,Intercept]
##          0.9622555      1.0673472      1.3589549      0.8415543

```

```

## r_pond[51,Intercept] r_pond[52,Intercept] r_pond[53,Intercept] r_pond[54,Intercept]
##          0.9910013      0.9237361      1.2107973      1.1505275
## r_pond[55,Intercept] r_pond[56,Intercept] r_pond[57,Intercept] r_pond[58,Intercept]
##          0.8304270      1.0743924      1.1347820      0.8837469
## r_pond[59,Intercept] r_pond[60,Intercept]
##          0.8612560      0.9822023      lp__
##          0.1988217

```

The `brms::neff_ratio()` function returns ratios of the effective samples over the total number of post-warmup iterations. So if we know the `neff_ratio()` values and the number of post-warmup iterations, the ‘Eff.Sample’ values are just a little algebra away. A quick solution is to look at the ‘total post-warmup samples’ line at the top of our `print()` output. Another way is to extract that information from our `brm()` fit object. I’m not aware of a way to do that directly, but we can extract the `iter` value (i.e., `b12.2$fit@sim$iter`), the `warmup` value (i.e., `b12.2$fit@sim$warmup`), and the number of `chains` (i.e., `b12.2$fit@sim$chains`). With those values in hand, simple algebra will return the ‘total post-warmup samples’ value. E.g.,

```
(n_iter <- (b12.3$fit@sim$iter - b12.3$fit@sim$warmup) * b12.3$fit@sim$chains)
```

```
## [1] 9000
```

And now we have `n_iter`, we can calculate the ‘Eff.Sample’ values.

```

neff_ratio(b12.3) %>%
  data.frame() %>%
  rownames_to_column() %>%
  set_names("parameter", "neff_ratio") %>%
  mutate(eff_sample = (neff_ratio * n_iter) %>% round(digits = 0)) %>%
  head()

```

```

##           parameter neff_ratio eff_sample
## 1      b_Intercept  0.3328983    2996
## 2 sd_pond__Intercept  0.3276092    2948
## 3 r_pond[1,Intercept]  1.3086470   11778
## 4 r_pond[2,Intercept]  1.4222957   12801
## 5 r_pond[3,Intercept]  1.7287578   15559
## 6 r_pond[4,Intercept]  1.7931600   16138

```

Digressions aside, let’s get ready for the diagnostic plot of Figure 12.3.

```

dsim %>%
  glimpse()

## #> Observations: 60
## #> Variables: 5
## #> $ pond     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...
## #> $ ni       <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ...
## #> $ true_a   <dbl> -0.82085139, 3.76575421, -0.03511672, 0.01999213, -1.59646315, 0.99155593, 0.9...
## #> $ si        <int> 0, 5, 4, 3, 0, 5, 5, 3, 5, 5, 3, 3, 3, 4, 4, 6, 10, 9, 9, 9, 10, 10, 6, 3, ...
## #> $ p_nopool <dbl> 0.00, 1.00, 0.80, 0.60, 0.00, 1.00, 0.60, 1.00, 1.00, 0.60, 0.60, 0.60, ...

# we could have included this step in the block of code below, if we wanted to
p_partpool <-
  coef(b12.3)$pond[, , ] %>%
  as_tibble() %>%
  transmute(p_partpool = inv_logit_scaled(Estimate))

dsim <-
  dsim %>%
  bind_cols(p_partpool) %>%

```

```

  mutate(p_true      = inv_logit_scaled(true_a)) %>%
  mutate(nopool_error = abs(p_nopool - p_true),
         partpool_error = abs(p_partpool - p_true))

dsim %>%
  glimpse()

```

Observations: 60
Variables: 9
\$ pond <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2...
\$ ni <int> 5, 10, 10, 10, 10, 10,...
\$ true_a <dbl> -0.82085139, 3.76575421, -0.03511672, 0.01999213, -1.59646315, 0.9915559...
\$ si <int> 0, 5, 4, 3, 0, 5, 5, 3, 5, 5, 3, 3, 3, 4, 4, 6, 10, 9, 9, 9, 10, 10, ...
\$ p_nopool <dbl> 0.00, 1.00, 0.80, 0.60, 0.00, 1.00, 1.00, 0.60, 1.00, 1.00, 0.60, 0.60, ...
\$ p_partpool <dbl> 0.2548519, 0.9089353, 0.8096274, 0.6809599, 0.2541776, 0.9097880, 0.9095...
\$ p_true <dbl> 0.3055830, 0.9773737, 0.4912217, 0.5049979, 0.1684765, 0.7293951, 0.7164...
\$ nopool_error <dbl> 0.305582963, 0.022626343, 0.308778278, 0.095002134, 0.168476520, 0.27060...
\$ partpool_error <dbl> 0.050731032, 0.068438323, 0.318405689, 0.175962060, 0.085701039, 0.18039...

Here is our code for Figure 12.3. The extra data processing for `dfline` is how we get the values necessary for the horizontal summary lines.

```

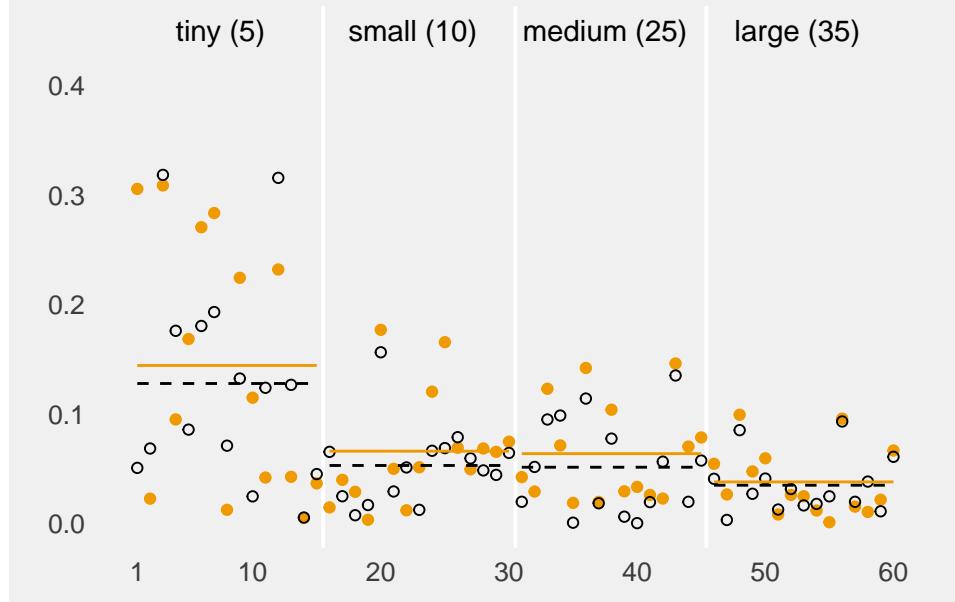
dfline <-
  dsim %>%
  select(ni, nopool_error:partpool_error) %>%
  gather(key, value, -ni) %>%
  group_by(key, ni) %>%
  summarise(mean_error = mean(value)) %>%
  mutate(x      = c( 1, 16, 31, 46),
        xend = c(15, 30, 45, 60))

dsim %>%
  ggplot(aes(x = pond)) +
  geom_vline(xintercept = c(15.5, 30.5, 45.4),
             color = "white", size = 2/3) +
  geom_point(aes(y = nopool_error), color = "orange2") +
  geom_point(aes(y = partpool_error), shape = 1) +
  geom_segment(data = dfline,
               aes(x = x, xend = xend,
                   y = mean_error, yend = mean_error),
               color = rep(c("orange2", "black"), each = 4),
               linetype = rep(1:2, each = 4)) +
  scale_x_continuous(breaks = c(1, 10, 20, 30, 40, 50, 60)) +
  annotate("text", x = c(15 - 7.5, 30 - 7.5, 45 - 7.5, 60 - 7.5), y = .45,
          label = c("tiny (5)", "small (10)", "medium (25)", "large (35)")) +
  labs(y      = "absolute error",
       title   = "Estimate error by model type",
       subtitle = "The horizontal axis displays pond number. The vertical axis measures\nthe absolute error in
theme_fivethirtyeight() +
theme(panel.grid    = element_blank(),
     plot.subtitle = element_text(size = 10))

```

Estimate error by model type

The horizontal axis displays pond number. The vertical axis measures the absolute error in the predicted proportion of survivors, compared to the true value used in the simulation. The higher the point, the worse the estimate. No-pooling shown in orange. Partial pooling shown in black. The orange and dashed black lines show the average error for each kind of estimate, across each initial density of tadpoles (pond size). Smaller ponds produce more error, but the partial pooling estimates are better on average, especially in smaller ponds.



If you wanted to quantify the difference in simple summaries, you might do something like this:

```
dsim %>%
  select(ni, nopool_error:partpool_error) %>%
  gather(key, value, -ni) %>%
  group_by(key) %>%
  summarise(mean_error = mean(value) %>% round(digits = 3),
            median_error = median(value) %>% round(digits = 3))
```

```
## # A tibble: 2 x 3
##   key      mean_error median_error
##   <chr>     <dbl>        <dbl>
## 1 nopool_error 0.078       0.05
## 2 partpool_error 0.067       0.051
```

I originally learned about the multilevel in order to work with [longitudinal data](#). In that context, I found the basic principles of a multilevel structure quite intuitive. The concept of partial pooling, however, took me some time to wrap my head around. If you're struggling with this, be patient and keep chipping away.

When McElreath [lectured on this topic in 2015](#), he traced partial pooling to statistician [Charles M. Stein](#). In 1977, Efron and Morris wrote the now classic paper, [Stein's Paradox in Statistics](#), which does a nice job breaking down why partial pooling can be so powerful. One of the primary examples they used in the paper was of 1970 batting average data. If you'd like more practice seeing how partial pooling works—or if you just like baseball—, check out my blog post, [Stein's Paradox and What Partial Pooling Can Do For You](#).

12.2.5.1 Overthinking: Repeating the pond simulation.

Within the brms workflow, we can reuse a compiled model with `update()`. But first, we'll simulate new data.

```

a      <- 1.4
sigma <- 1.5
n_ponds <- 60

set.seed(1999) # for new data, set a new seed
new_dsim <-
  tibble(pond = 1:n_ponds,
         ni = rep(c(5, 10, 25, 35), each = n_ponds / 4) %>% as.integer(),
         true_a = rnorm(n = n_ponds, mean = a, sd = sigma)) %>%
  mutate(si = rbinom(n = n(), prob = inv_logit_scaled(true_a), size = ni)) %>%
  mutate(p_nopool = si / ni)

glimpse(new_dsim)

## Observations: 60
## Variables: 5
## $ pond    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...
## $ ni      <int> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 10, 10, 10, 10, 10, 10, 10, 1...
## $ true_a  <dbl> 2.4990087, 1.3432554, 3.2045137, 3.6047030, 1.6005354, 2.1797409, 0.5759270, ...
## $ si      <int> 4, 4, 5, 4, 4, 2, 4, 3, 5, 4, 5, 2, 2, 5, 10, 7, 10, 10, 8, 10, 9, 5, 10, 1...
## $ p_nopool <dbl> 0.80, 0.80, 1.00, 0.80, 0.80, 0.40, 0.80, 0.60, 1.00, 0.80, 1.00, 0.40, ...

```

Fit the new model.

```

b12.3_new <-
  update(b12.3,
         newdata = new_dsim,
         iter = 10000, warmup = 1000, chains = 1, cores = 1)

```

```
print(b12.3_new)
```

```

## Family: binomial
##   Links: mu = logit
## Formula: si | trials(ni) ~ 1 + (1 | pond)
##   Data: new_dsim (Number of observations: 60)
## Samples: 1 chains, each with iter = 10000; warmup = 1000; thin = 1;
##           total post-warmup samples = 9000
##
## Group-Level Effects:
##   ~pond (Number of levels: 60)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##   sd(Intercept)     1.26      0.18      0.95      1.66        3633 1.00
## 
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##   Intercept       1.53      0.20      1.16      1.92        3279 1.00
## 
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Why not plot the first simulation versus the second one?

```

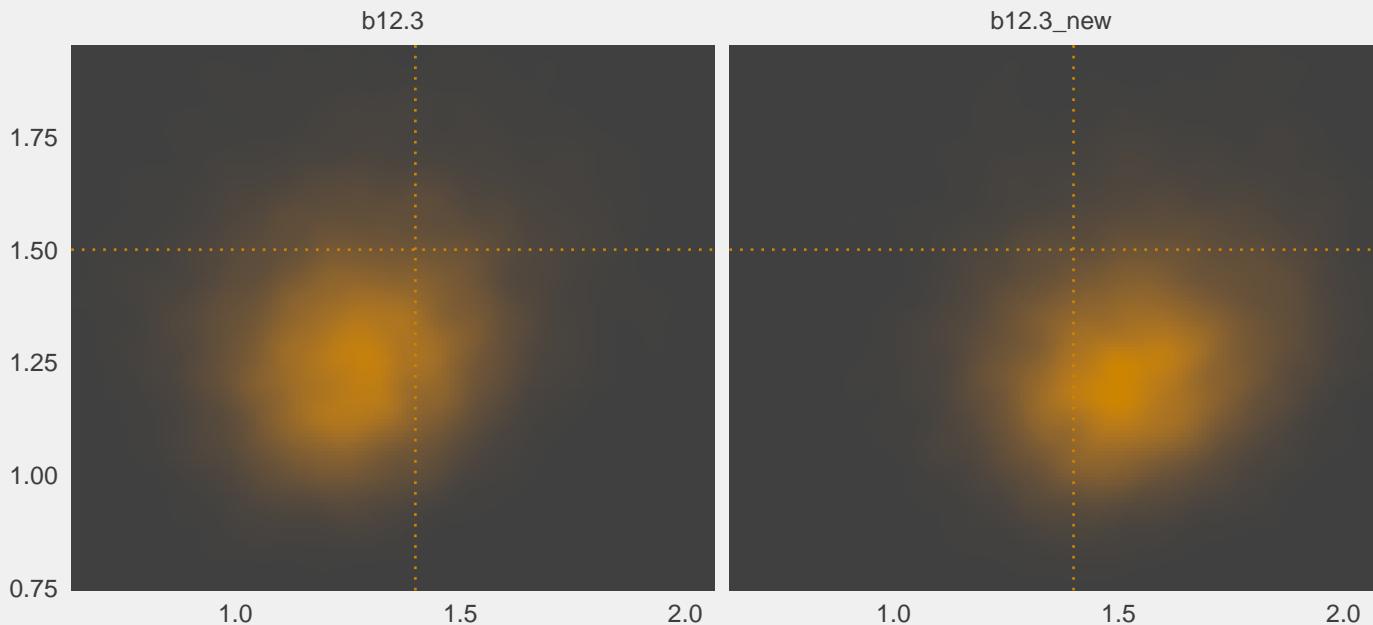
bind_rows(posterior_samples(b12.3),
          posterior_samples(b12.3_new)) %>%
  mutate(model = rep(c("b12.3", "b12.3_new"), each = n() / 2)) %>%

```

```
ggplot(aes(x = b_Intercept, y = sd_pond_Intercept)) +
  stat_density_2d(geom = "raster",
                  aes(fill = stat(density)),
                  contour = F, n = 200) +
  geom_vline(xintercept = a, color = "orange3", linetype = 3) +
  geom_hline(yintercept = sigma, color = "orange3", linetype = 3) +
  scale_fill_gradient(low = "grey25", high = "orange3") +
  ggtitle("Our simulation posteriors contrast a bit",
          subtitle = expression(paste(alpha, " is on the x and ", sigma, " is on the y, both in log-odds. The ")),
  coord_cartesian(xlim = c(.7, 2),
                  ylim = c(.8, 1.9)) +
  theme_fivethirtyeight() +
  theme(legend.position = "none",
        panel.grid      = element_blank()) +
  facet_wrap(~model, ncol = 2)
```

Our simulation posteriors contrast a bit

α is on the x and σ is on the y, both in log-odds. The dotted lines intersect at the true values.



If you'd like the `stanfit` portion of your `brm()` object, subset with `$fit`. Take `b12.3`, for example. You might check out its structure via `b12.3$fit %>% str()`. Here's the actual Stan code.

```
b12.3$fit@ stanmodel
```

```
## S4 class stanmodel 'e57042e640ec9cd4c321d064bf9ac5e3' coded as follows:
## // generated with brms 2.8.0
## functions {
## }
## data {
##   int<lower=1> N; // number of observations
##   int Y[N]; // response variable
##   int trials[N]; // number of trials
##   // data for group-level effects of ID 1
##   int<lower=1> N_1;
##   int<lower=1> M_1;
##   int<lower=1> J_1[N];
```

```

##  vector[N] Z_1_1;
##  int prior_only; // should the likelihood be ignored?
## }
## transformed data {
## }
## parameters {
##   real temp_Intercept; // temporary intercept
##   vector<lower=0>[M_1] sd_1; // group-level standard deviations
##   vector[N_1] z_1[M_1]; // unscaled group-level effects
## }
## transformed parameters {
##   // group-level effects
##   vector[N_1] r_1_1 = (sd_1[1] * (z_1[1]));
## }
## model {
##   vector[N] mu = temp_Intercept + rep_vector(0, N);
##   for (n in 1:N) {
##     mu[n] += r_1_1[J_1[n]] * Z_1_1[n];
##   }
##   // priors including all constants
##   target += normal_lpdf(temp_Intercept | 0, 1);
##   target += cauchy_lpdf(sd_1 | 0, 1)
##     - 1 * cauchy_lccdf(0 | 0, 1);
##   target += normal_lpdf(z_1[1] | 0, 1);
##   // likelihood including all constants
##   if (!prior_only) {
##     target += binomial_logit_lpmf(Y | trials, mu);
##   }
## }
## generated quantities {
##   // actual population-level intercept
##   real b_Intercept = temp_Intercept;
## }
##

```

And you can get the data of a given `brm()` fit object like so.

```
b12.3$data %>%
  head()
```

```

##  si ni pond
## 1 0 5 1
## 2 5 5 2
## 3 4 5 3
## 4 3 5 4
## 5 0 5 5
## 6 5 5 6

```

12.3 More than one type of cluster

“We can use and often should use more than one type of cluster in the same model” (p. 370).

12.3.1 Multilevel chimpanzees.

The initial multilevel update from model `b10.4` from the last chapter follows the statistical formula

$$\begin{aligned}
 \text{left_pull}_i &\sim \text{Binomial}(n_i = 1, p_i) \\
 \text{logit}(p_i) &= \alpha + \alpha_{\text{actor}_i} + (\beta_1 + \beta_2 \text{condition}_i) \text{prosoc_left}_i \\
 \alpha_{\text{actor}} &\sim \text{Normal}(0, \sigma_{\text{actor}}) \\
 \alpha &\sim \text{Normal}(0, 10) \\
 \beta_1 &\sim \text{Normal}(0, 10) \\
 \beta_2 &\sim \text{Normal}(0, 10) \\
 \sigma_{\text{actor}} &\sim \text{HalfCauchy}(0, 1)
 \end{aligned}$$

Notice that α is inside the linear model, not inside the Gaussian prior for α_{actor} . This is mathematically equivalent to what [we] did with the tadpoles earlier in the chapter. You can always take the mean out of a Gaussian distribution and treat that distribution as a constant plus a Gaussian distribution centered on zero.

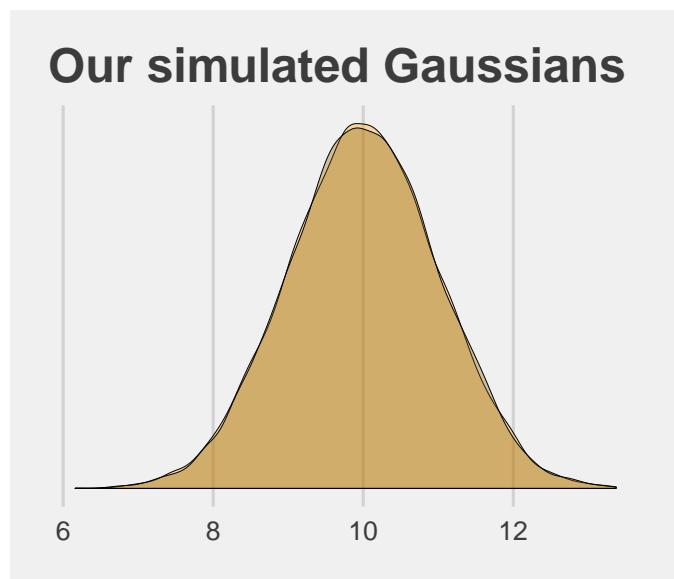
This might seem a little weird at first, so it might help train your intuition by experimenting in R. (p. 371)

Behold our two identical Gaussians in a tidy tibble.

```
set.seed(12)
two_gaussians <-
  tibble(y1 =      rnorm(n = 1e4, mean = 10, sd = 1),
        y2 = 10 + rnorm(n = 1e4, mean = 0, sd = 1))
```

Let's follow McElreath's advice to make sure they are same by superimposing the density of one on the other.

```
two_gaussians %>%
  ggplot() +
  geom_density(aes(x = y1),
               size = 0, fill = "orange1", alpha = 1/3) +
  geom_density(aes(x = y2),
               size = 0, fill = "orange4", alpha = 1/3) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(title = "Our simulated Gaussians") +
  theme_fivethirtyeight()
```



Yep, those Gaussians look about the same.

Let's get the `chimpanzees` data from rethinking.

```
library(rethinking)
data(chimpanzees)
d <- chimpanzees
```

Detach rethinking and reload brms.

```
rm(chimpanzees)
detach(package:rethinking, unload = T)
library(brms)
```

For our brms model with varying intercepts for `actor` but not `block`, we employ the `pulled_left ~ 1 + ... + (1 | actor)` syntax, specifically omitting a `(1 | block)` section.

```
b12.4 <-
  brm(data = d, family = binomial,
       pulled_left | trials(1) ~ 1 + prosoc_left + prosoc_left:condition + (1 | actor),
       prior = c(prior(normal(0, 10), class = Intercept),
                  prior(normal(0, 10), class = b),
                  prior(cauchy(0, 1), class = sd)),
       # I'm using 4 cores, instead of the `cores=3` in McElreath's code
       iter = 5000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = 0.95),
       seed = 12)
```

The initial solutions came with a few divergent transitions. Increasing `adapt_delta` to 0.95 solved the problem. You can also solve the problem with more strongly regularizing priors such as `normal(0, 2)` on the intercept and slope parameters (see [recommendations from the Stan team](#)). Consider trying both methods and comparing the results. They're similar.

Here we add the `actor`-level deviations to the fixed intercept, the grand mean.

```
post <- posterior_samples(b12.4)

post %>%
  select(starts_with("r_actor")) %>%
  gather() %>%
  # this is how we might add the grand mean to the actor-level deviations
  mutate(value = value + post$b_Intercept) %>%
  group_by(key) %>%
  summarise(mean = mean(value) %>% round(digits = 2))
```

```
## # A tibble: 7 x 2
##   key                 mean
##   <chr>              <dbl>
## 1 r_actor[1,Intercept] -0.71
## 2 r_actor[2,Intercept]  4.6 
## 3 r_actor[3,Intercept] -1.02
## 4 r_actor[4,Intercept] -1.02
## 5 r_actor[5,Intercept] -0.71
## 6 r_actor[6,Intercept]  0.23
## 7 r_actor[7,Intercept]  1.76
```

Here's another way to get at the same information, this time using `coef()` and a little formatting help from the `stringr::str_c()` function. Just for kicks, we'll throw in the 95% intervals, too.

```
coef(b12.4)$actor[, c(1, 3:4), 1] %>%
  as_tibble() %>%
  round(digits = 2) %>%
  # here we put the credible intervals in an APA-6-style format
```

```

mutate(`95% CIs` = str_c("[", Q2.5, ", ", Q97.5, "]"),
       actor      = str_c("chimp #", 1:7)) %>%
rename(mean = Estimate) %>%
select(actor, mean, `95% CIs`) %>%
knitr::kable()

```

actor	mean	95% CIs
chimp #1	-0.71	[-1.24, -0.2]
chimp #2	4.60	[2.54, 8.49]
chimp #3	-1.02	[-1.57, -0.48]
chimp #4	-1.02	[-1.57, -0.49]
chimp #5	-0.71	[-1.23, -0.21]
chimp #6	0.23	[-0.29, 0.77]
chimp #7	1.76	[1.06, 2.54]

If you prefer the posterior median to the mean, just add a `robust = T` argument inside the `coef()` function.

12.3.2 Two types of cluster.

The full statistical model follows the form

$$\begin{aligned}
\text{left_pull}_i &\sim \text{Binomial}(n_i = 1, p_i) \\
\logit(p_i) &= \alpha + \alpha_{\text{actor}_i} + \alpha_{\text{block}_i} + (\beta_1 + \beta_2 \text{condition}_i) \text{prosoc_left}_i \\
\alpha_{\text{actor}} &\sim \text{Normal}(0, \sigma_{\text{actor}}) \\
\alpha_{\text{block}} &\sim \text{Normal}(0, \sigma_{\text{actor}}) \\
\alpha &\sim \text{Normal}(0, 10) \\
\beta_1 &\sim \text{Normal}(0, 10) \\
\beta_2 &\sim \text{Normal}(0, 10) \\
\sigma_{\text{actor}} &\sim \text{HalfCauchy}(0, 1) \\
\sigma_{\text{block}} &\sim \text{HalfCauchy}(0, 1)
\end{aligned}$$

Our brms model with varying intercepts for both `actor` and `block` now employs the `... (1 | actor) + (1 | block)` syntax.

```

b12.5 <-
  update(b12.4,
    newdata = d,
    formula = pulled_left | trials(1) ~ 1 + prosoc_left + prosoc_left:condition +
      (1 | actor) + (1 | block),
    iter = 6000, warmup = 1000, cores = 4, chains = 4,
    control = list(adapt_delta = 0.99),
    seed = 12)

```

This time we increased `adapt_delta` to 0.99 to avoid divergent transitions. We can look at the primary coefficients with `print()`. McElreath encouraged us to inspect the trace plots. Here they are.

```

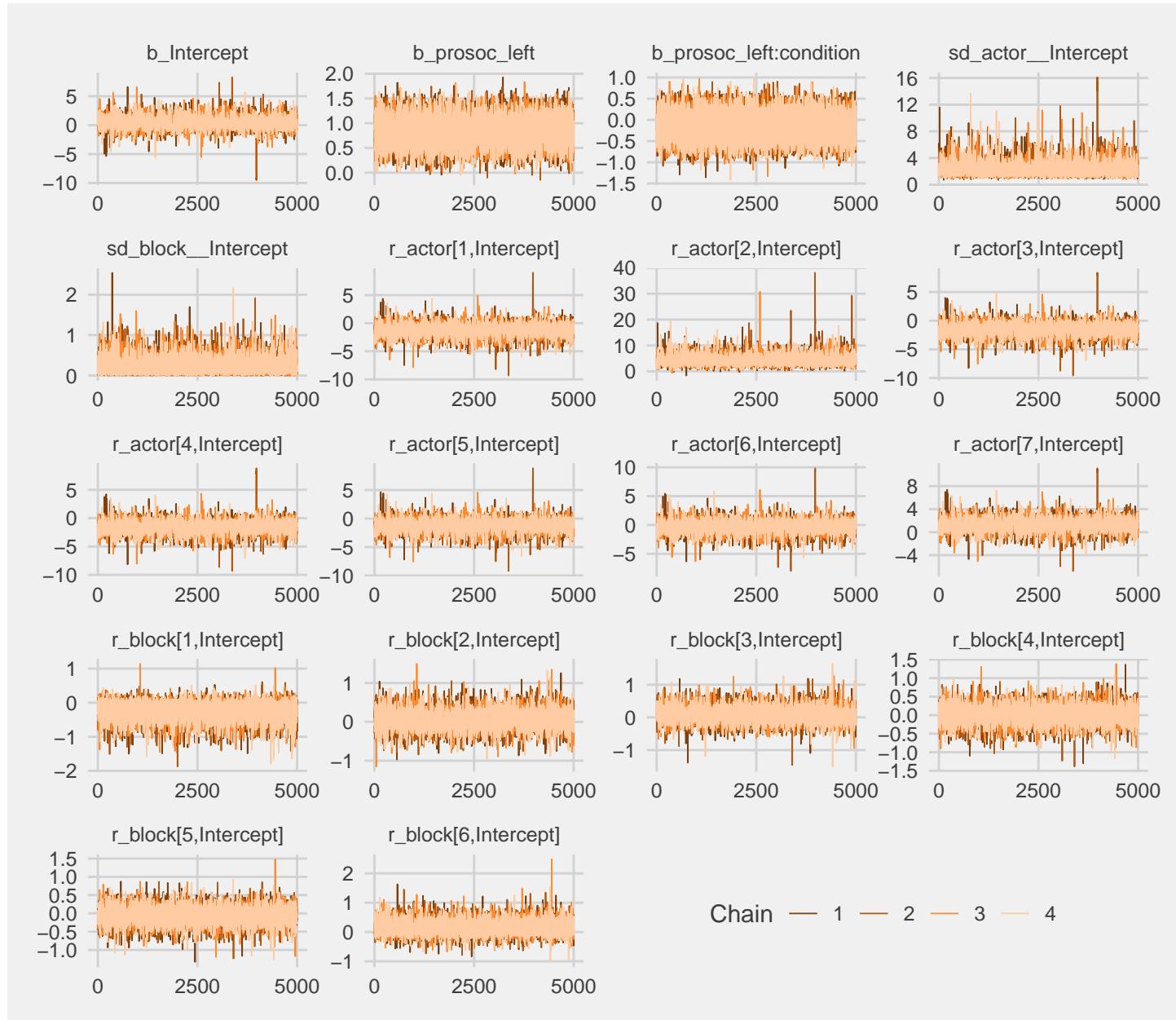
library(bayesplot)
color_scheme_set("orange")

post <- posterior_samples(b12.5, add_chain = T)

post %>%
  select(-lp__, -iter) %>%
  mcmc_trace(facet_args = list(ncol = 4)) +
  scale_x_continuous(breaks = c(0, 2500, 5000)) +

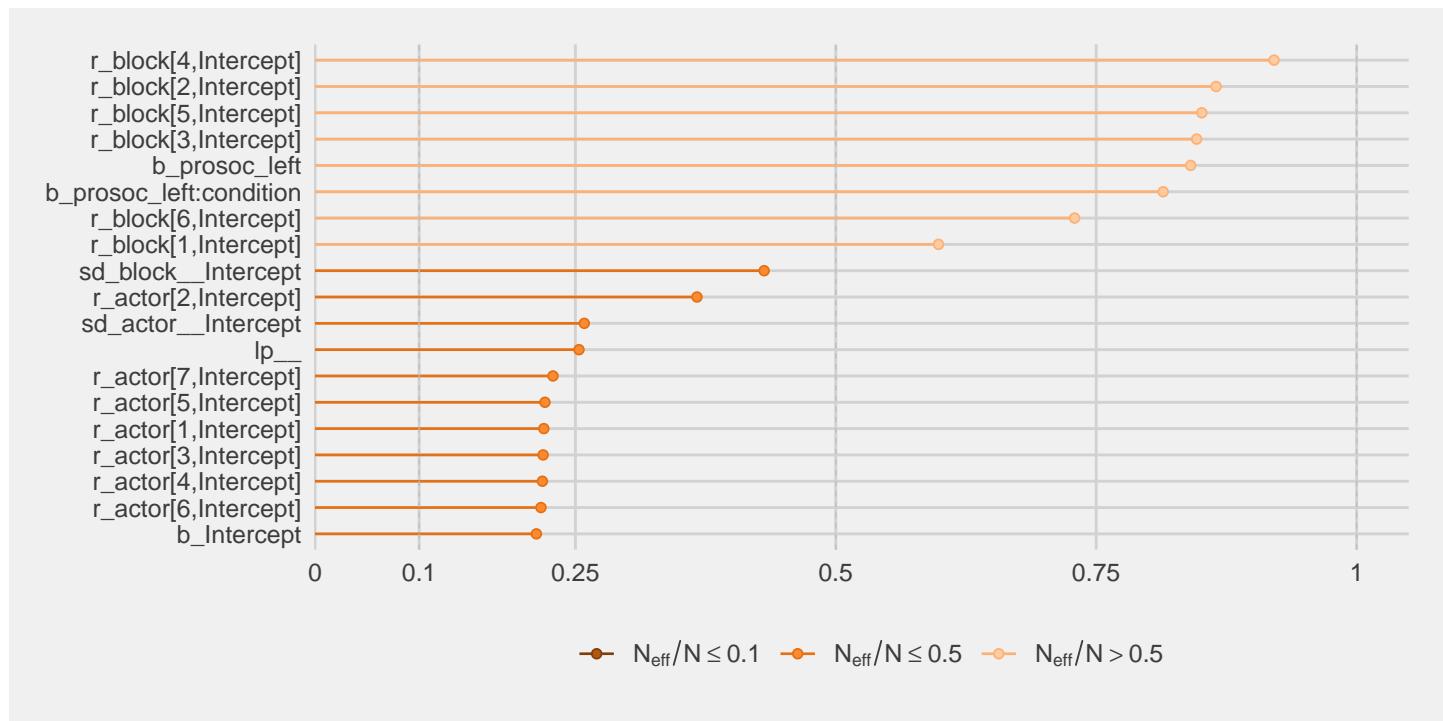
```

```
theme_fivethirtyeight() +
theme(legend.position = c(.75, .06))
```



The trace plots look great. We may as well examine the n_{eff}/N ratios, too.

```
neff_ratio(b12.5) %>%
mcmc_neff() +
theme_fivethirtyeight()
```



About half of them are lower than we might like, but none are in the embarrassing $n_{\text{eff}}/N \leq .1$ range. Let's look at the summary of the main parameters.

```
print(b12.5)
```

```
## Family: binomial
## Links: mu = logit
## Formula: pulled_left | trials(1) ~ prosoc_left + (1 | actor) + (1 | block) + prosoc_left:condition
## Data: d (Number of observations: 504)
## Samples: 4 chains, each with iter = 6000; warmup = 1000; thin = 1;
##          total post-warmup samples = 20000
##
## Group-Level Effects:
## ~actor (Number of levels: 7)
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)    2.27      0.97    1.13     4.63       5171  1.00
## 
## ~block (Number of levels: 6)
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)    0.22      0.18     0.01     0.66       8624  1.00
## 
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
## Intercept        0.43      0.99   -1.39     2.47       4251  1.00
## prosoc_left      0.83      0.26     0.31     1.35      16813  1.00
## prosoc_left:condition -0.14      0.30   -0.73     0.45      16285  1.00
## 
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

This time, we'll need to use `brms::ranef()` to get those depth=2-type estimates in the same metric displayed in the text. With `ranef()`, you get the group-specific estimates in a deviance metric. The `coef()` function, in contrast, yields the group-specific estimates in what you might call the natural metric. We'll get more language for this in the next chapter.

```
ranef(b12.5)$actor[, , "Intercept"] %>%
  round(digits = 2)
```

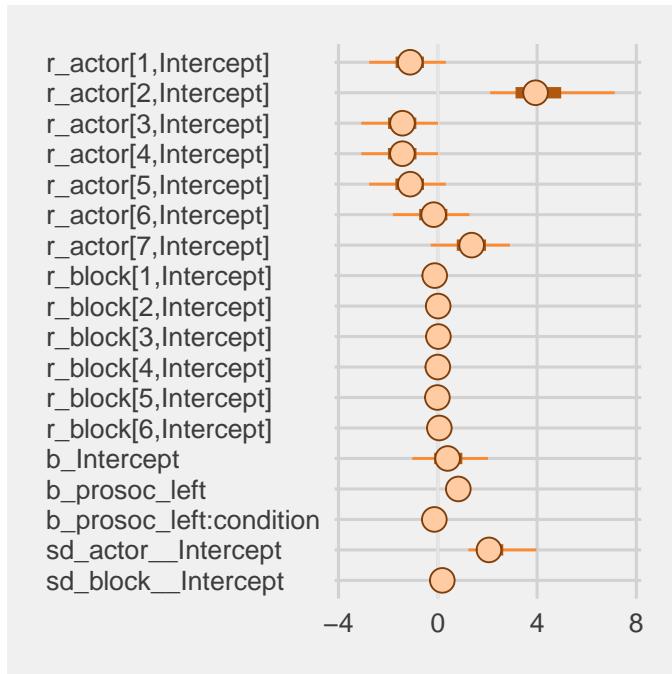
	## Estimate	Est.Error	Q2.5	Q97.5
## 1	-1.15	1.00	-3.24	0.68
## 2	4.21	1.72	1.78	8.14
## 3	-1.46	1.00	-3.54	0.37
## 4	-1.46	1.00	-3.54	0.36
## 5	-1.15	1.00	-3.21	0.65
## 6	-0.20	1.00	-2.29	1.64
## 7	1.34	1.02	-0.74	3.24

```
ranef(b12.5)$block[, , "Intercept"] %>%
  round(digits = 2)
```

	## Estimate	Est.Error	Q2.5	Q97.5
## 1	-0.18	0.23	-0.74	0.12
## 2	0.04	0.19	-0.33	0.45
## 3	0.05	0.19	-0.30	0.48
## 4	0.00	0.18	-0.38	0.40
## 5	-0.04	0.18	-0.46	0.32
## 6	0.11	0.20	-0.21	0.59

We might make the coefficient plot of Figure 12.4.a like this:

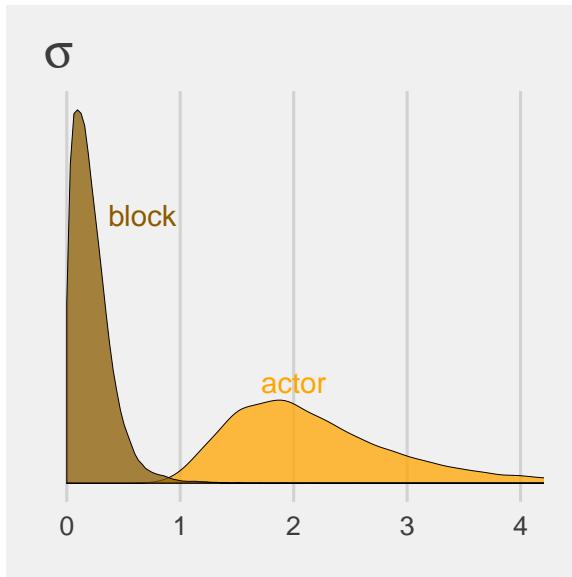
```
stanplot(b12.5, pars = c("^r_",
  "b_",
  "sd_")) +
  theme_fivethirtyeight() +
  theme(axis.text.y = element_text(hjust = 0))
```



Once we get the posterior samples, it's easy to compare the random variances as in Figure 12.4.b.

```
post %>%
  ggplot(aes(x = sd_actor__Intercept)) +
  geom_density(size = 0, fill = "orange1", alpha = 3/4) +
  geom_density(aes(x = sd_block__Intercept),
               size = 0, fill = "orange4", alpha = 3/4) +
```

```
scale_y_continuous(NULL, breaks = NULL) +
coord_cartesian(xlim = c(0, 4)) +
labs(title = expression(sigma)) +
annotate("text", x = 2/3, y = 2, label = "block", color = "orange4") +
annotate("text", x = 2, y = 3/4, label = "actor", color = "orange1") +
theme_fivethirtyeight()
```



We might compare our models by their PSIS-LOO values.

```
b12.4 <- add_criterion(b12.4, "loo")
b12.5 <- add_criterion(b12.5, "loo")

loo_compare(b12.4, b12.5) %>%
  print(simplify = F)

##      elpd_diff se_diff elpd_loo se_elpd_loo p_loo    se_p_loo looic    se_looic
## b12.4     0.0      0.0 -265.7      9.8     8.1      0.4   531.4     19.5
## b12.5    -0.6      0.9 -266.3      9.9    10.4      0.5   532.6     19.7
```

The two models yield nearly-equivalent information criteria values. Yet recall what McElreath wrote: “There is nothing to gain here by selecting either model. The comparison of the two models tells a richer story” (p. 367).

12.4 Multilevel posterior predictions

... producing implied predictions from a fit model, is very helpful for understanding what the model means. Every model is a merger of sense and nonsense. When we understand a model, we can find its sense and control its nonsense. But as models get more complex, it is very difficult to impossible to understand them just by inspecting tables of posterior means and intervals. Exploring implied posterior predictions helps much more...

... The introduction of varying effects does introduce nuance, however.

First, we should no longer expect the model to exactly retrodict the sample, because adaptive regularization has as its goal to trade off poorer fit in sample for better inference and hopefully better fit out of sample. This is what shrinkage does for us...

Second, “prediction” in the context of a multilevel model requires additional choices. If we wish to validate a model against the specific clusters used to fit the model, that is one thing. But if we instead wish to compute predictions for new clusters, other than the one observed in the sample, that is quite another. We’ll consider each of these in turn, continuing to use the chimpanzees model from the previous section. (p. 376)

12.4.1 Posterior prediction for same clusters.

Like McElreath did in the text, we'll do this two ways. Recall we use `brms::fitted()` in place of `rethinking::link()`.

```
chimp <- 2
nd <-
  tibble(prosoc_left = c(0, 1, 0, 1),
         condition = c(0, 0, 1, 1),
         actor      = chimp)

(
  chimp_2_fitted <-
    fitted(b12.4,
           newdata = nd) %>%
    as_tibble() %>%
    mutate(condition = factor(c("0/0", "1/0", "0/1", "1/1"),
                               levels = c("0/0", "1/0", "0/1", "1/1")))
)

## # A tibble: 4 x 5
##   Estimate Est.Error Q2.5 Q97.5 condition
##     <dbl>     <dbl> <dbl> <dbl> <fct>
## 1     0.980    0.0198  0.927  1.000  0/0
## 2     0.991    0.00963  0.965  1.000  1/0
## 3     0.980    0.0198  0.927  1.000  0/1
## 4     0.990    0.0109   0.960  1.000  1/1

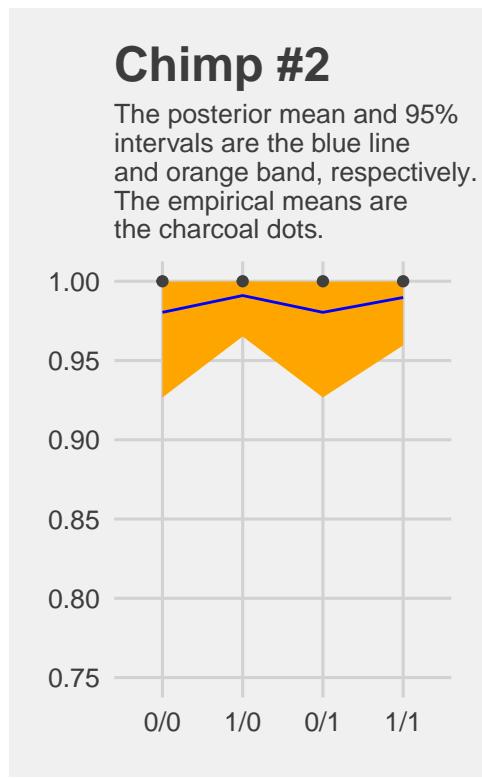
(
  chimp_2_d <-
  d %>%
  filter(actor == chimp) %>%
  group_by(prosoc_left, condition) %>%
  summarise(prob = mean(pulled_left)) %>%
  ungroup() %>%
  mutate(condition = str_c(prosoc_left, "/", condition)) %>%
  mutate(condition = factor(condition, levels = c("0/0", "1/0", "0/1", "1/1")))
)

## # A tibble: 4 x 3
##   prosoc_left condition  prob
##     <int> <fct>     <dbl>
## 1       0 0/0        1
## 2       0 0/1        1
## 3       1 1/0        1
## 4       1 1/1        1
```

McElreath didn't show the corresponding plot in the text. It might look like this.

```
chimp_2_fitted %>%
  # if you want to use `geom_line()` or `geom_ribbon()` with a factor on the x axis,
  # you need to code something like `group = 1` in `aes()`
  ggplot(aes(x = condition, y = Estimate, group = 1)) +
  geom_ribbon(aes(ymin = Q2.5, ymax = Q97.5), fill = "orange1") +
  geom_line(color = "blue") +
  geom_point(data = chimp_2_d,
             aes(y = prob),
             color = "grey25") +
  ggtitle("Chimp #2",
```

```
subtitle = "The posterior mean and 95%\nintervals are the blue line\nand orange band, respectively.\n\ncoord_cartesian(ylim = c(.75, 1)) +\ntheme_fivethirtyeight() +\ntheme(plot.subtitle = element_text(size = 10))
```



Do note how severely we've restricted the y-axis range. But okay, now let's do things by hand. We'll need to extract the posterior samples and look at the structure of the data.

```
post <- posterior_samples(b12.4)

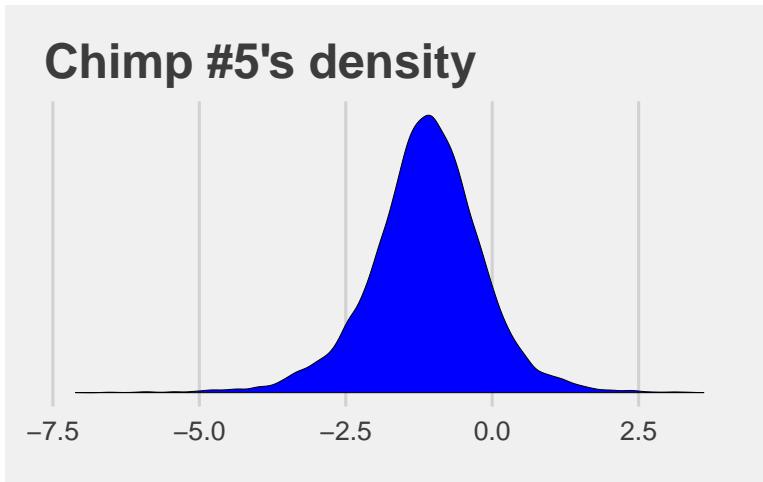
glimpse(post)

## Observations: 16,000
## Variables: 12
## $ b_Intercept           <dbl> -0.49845428, 0.36263851, 1.91766750, 1.82740632, 2.24556133, ...
## $ b_prosoc_left          <dbl> 0.9032174, 1.3799669, 0.8727582, 0.8015609, 0.6627169, 1.1561...
## $ `b_prosoc_left:condition` <dbl> -0.56182114, -0.46722870, -0.70816999, -0.33963142, -0.492577...
## $ sd_actor__Intercept    <dbl> 2.571271, 1.476708, 2.084258, 2.729417, 2.724787, 2.071737, 1...
## $ `r_actor[1,Intercept]` <dbl> -0.468006872, -1.364675261, -2.403394919, -2.574390330, -2.79...
## $ `r_actor[2,Intercept]` <dbl> 5.533991, 2.910396, 5.437039, 4.342981, 3.537432, 2.313620, 3...
## $ `r_actor[3,Intercept]` <dbl> -0.45618115, -1.59925747, -2.81698107, -2.51778170, -3.157804...
## $ `r_actor[4,Intercept]` <dbl> -0.65347198, -1.77826745, -2.54650655, -2.79884389, -2.961426...
## $ `r_actor[5,Intercept]` <dbl> -0.3303643, -1.3395610, -2.4678893, -2.5197585, -2.6281694, ...
## $ `r_actor[6,Intercept]` <dbl> 0.56380900, -0.67941570, -1.10245504, -1.98517600, -1.5241480...
## $ `r_actor[7,Intercept]` <dbl> 2.0842292, 1.6935043, -0.5287177, 0.4990477, -0.4595667, 1.02...
## $ lp_<...>
```

McElreath didn't show what his R code 12.29 `dens(post$a_actor[,5])` would look like. But here's our analogue.

```
post %>%
  transmute(actor_5 = `r_actor[5,Intercept]`) %>%
  ggplot(aes(x = actor_5)) +
```

```
geom_density(size = 0, fill = "blue") +
scale_y_continuous(breaks = NULL) +
ggtitle("Chimp #5's density") +
theme_fivethirtyeight()
```



And because we made the density only using the `r_actor[5,Intercept]` values (i.e., we didn't add `b_Intercept` to them), the density is in a deviance-score metric.

McElreath built his own `link()` function. Here we'll build an alternative to `fitted()`.

```
# our hand-made `brms::fitted()` alternative
my_fitted <- function(prosoc_left, condition){
  post %>%
    transmute(fitted = (b_Intercept +
      `r_actor[5,Intercept]` +
      b_prosoc_left * prosoc_left +
      `b_prosoc_left:condition` * prosoc_left * condition) %>%
    inv_logit_scaled())
}

# the posterior summaries
(
  chimp_5_my_fitted <-
  tibble(prosoc_left = c(0, 1, 0, 1),
        condition     = c(0, 0, 1, 1)) %>%
  mutate(post = map2(prosoc_left, condition, my_fitted)) %>%
  unnest() %>%
  mutate(condition = str_c(prosoc_left, "/", condition)) %>%
  mutate(condition = factor(condition, levels = c("0/0", "1/0", "0/1", "1/1"))) %>%
  group_by(condition) %>%
  tidybayes::mean_qi(fitted)
)

## # A tibble: 4 x 7
##   condition fitted .lower .upper .width .point .interval
##   <fct>     <dbl>  <dbl>  <dbl> <dbl> <chr>  <chr>
## 1 0/0       0.331  0.226  0.448  0.95 mean    qi
## 2 1/0       0.527  0.384  0.667  0.95 mean    qi
## 3 0/1       0.331  0.226  0.448  0.95 mean    qi
## 4 1/1       0.495  0.354  0.637  0.95 mean    qi

# the empirical summaries
chimp <- 5
```

```

(
  chimp_5_d <-
  d %>%
  filter(actor == chimp) %>%
  group_by(prosoc_left, condition) %>%
  summarise(prob = mean(pulled_left)) %>%
  ungroup() %>%
  mutate(condition = str_c(prosoc_left, "/", condition)) %>%
  mutate(condition = factor(condition, levels = c("0/0", "1/0", "0/1", "1/1")))
)

## # A tibble: 4 x 3
##   prosoc_left condition  prob
##       <int>     <fct>    <dbl>
## 1          0     0/0      0.333
## 2          0     0/1      0.278
## 3          1     1/0      0.556
## 4          1     1/1      0.5

```

Okay, let's see how good we are at retrodicting the `pulled_left` probabilities for `actor == 5`.

```

chimp_5_my_fitted %>%
  ggplot(aes(x = condition, y = fitted, group = 1)) +
  geom_ribbon(aes(ymin = .lower, ymax = .upper), fill = "orange1") +
  geom_line(color = "blue") +
  geom_point(data = chimp_5_d,
             aes(y = prob),
             color = "grey25") +
  ggtitle("Chimp #5",
          subtitle = "This plot is like the last except\nwe did more by hand.") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.subtitle = element_text(size = 10))

```

Chimp #5

This plot is like the last except
we did more by hand.



Not bad.

12.4.2 Posterior prediction for new clusters.

By average actor, McElreath referred to a chimp with an intercept exactly at the population mean α . So this time we'll only be working with the population parameters, or what are also sometimes called the fixed effects. When using `brms::posterior_samples()` output, this would mean working with columns beginning with the `b_` prefix (i.e., `b_Intercept`, `b_prosoc_left`, and `b_prosoc_left:condition`).

```
post_average_actor <-
  post %>%
    # here we use the linear regression formula to get the log_odds for the 4 conditions
    transmute(`0/0` = b_Intercept,
              `1/0` = b_Intercept + b_prosoc_left,
              `0/1` = b_Intercept,
              `1/1` = b_Intercept + b_prosoc_left + `b_prosoc_left:condition`) %>%
    # with `mutate_all()` we can convert the estimates to probabilities in one fell swoop
    mutate_all(inv_logit_scaled) %>%
    # putting the data in the long format and grouping by condition (i.e., `key`)
    gather() %>%
    mutate(key = factor(key, level = c("0/0", "1/0", "0/1", "1/1"))) %>%
    group_by(key) %>%
    # here we get the summary values for the plot
    summarise(m = mean(value),
              # note we're using 80% intervals
              ll = quantile(value, probs = .1),
              ul = quantile(value, probs = .9))

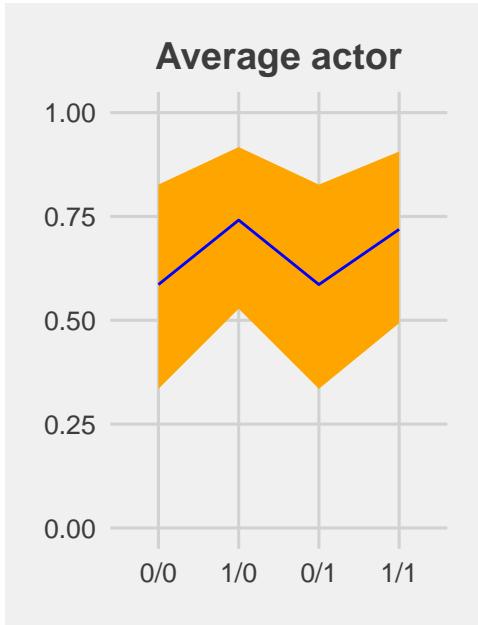
post_average_actor
```

```
## # A tibble: 4 x 4
##   key      m     ll     ul
##   <fct> <dbl> <dbl> <dbl>
## 1 0/0    0.586  0.335  0.827
## 2 1/0    0.741  0.528  0.917
## 3 0/1    0.586  0.335  0.827
## 4 1/1    0.719  0.493  0.906
```

Figure 12.5.a.

```
p1 <-
  post_average_actor %>%
  ggplot(aes(x = key, y = m, group = 1)) +
  geom_ribbon(aes(ymin = ll, ymax = ul), fill = "orange1") +
  geom_line(color = "blue") +
  ggtitle("Average actor") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 14, hjust = .5))
```

p1



If we want to depict the variability across the chimps, we need to include `sd_actor__Intercept` into the calculations. In the first block of code, below, we simulate a bundle of new intercepts defined by

$$\alpha_{\text{actor}} \sim \text{Normal}(0, \sigma_{\text{actor}})$$

```
# the random effects
set.seed(12.42)
ran_ef <-
  tibble(random_effect = rnorm(n = 1000, mean = 0, sd = post$sd_actor__Intercept)) %>%
  # with the `., ., ., .` syntax, we quadruple the previous line
  bind_rows(., ., ., .)

# the fixed effects (i.e., the population parameters)
fix_ef <-
  post %>%
  slice(1:1000) %>%
  transmute(`0/0` = b_Intercept,
           `1/0` = b_Intercept + b_prosoc_left,
           `0/1` = b_Intercept,
           `1/1` = b_Intercept + b_prosoc_left + `b_prosoc_left:condition`) %>%
gather() %>%
  rename(condition = key,
        fixed_effect = value) %>%
  mutate(condition = factor(condition, level = c("0/0", "1/0", "0/1", "1/1")))

# combine them
ran_and_fix_ef <-
  bind_cols(ran_ef, fix_ef) %>%
  mutate(intercept = fixed_effect + random_effect) %>%
  mutate(prob      = inv_logit_scaled(intercept))

# to simplify things, we'll reduce them to summaries
(
  marginal_effects <-
  ran_and_fix_ef %>%
  group_by(condition) %>%
  summarise(m = mean(prob),
            ll = quantile(prob, probs = .1),
```

```

    ul = quantile(prob, probs = .9))
}

## # A tibble: 4 x 4
##   condition     m     ll     ul
##   <fct>     <dbl>   <dbl>   <dbl>
## 1 0/0        0.559  0.0860  0.970
## 2 1/0        0.673  0.177   0.986
## 3 0/1        0.559  0.0860  0.970
## 4 1/1        0.657  0.163   0.984

```

Behold Figure 12.5.b.

```

p2 <-
  marginal_effects %>%
  ggplot(aes(x = condition, y = m, group = 1)) +
  geom_ribbon(aes(ymin = ll, ymax = ul), fill = "orange1") +
  geom_line(color = "blue") +
  ggtitle("Marginal of actor") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 14, hjust = .5))

```

p2

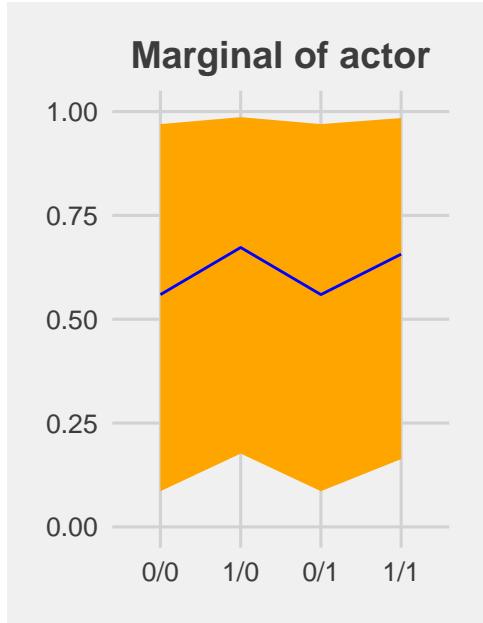


Figure 12.5.c just takes a tiny bit more wrangling.

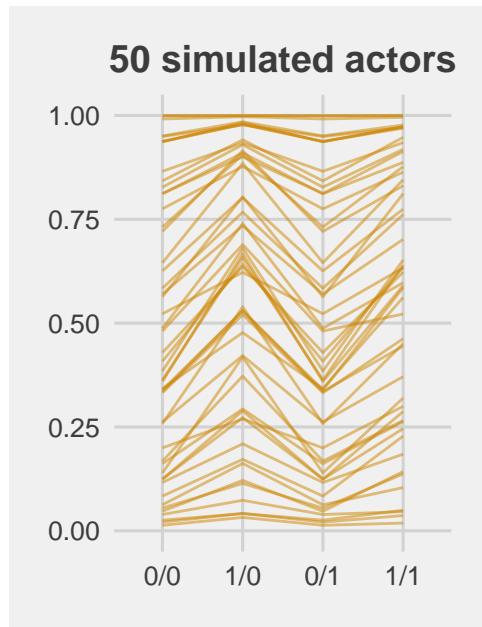
```

p3 <-
  ran_and_fix_ef %>%
  mutate(iter = rep(1:1000, times = 4)) %>%
  filter(iter %in% c(1:50)) %>%

  ggplot(aes(x = condition, y = prob, group = iter)) +
  theme_fivethirtyeight() +
  ggtitle("50 simulated actors") +
  coord_cartesian(ylim = 0:1) +
  geom_line(alpha = 1/2, color = "orange3") +
  theme(plot.title = element_text(size = 14, hjust = .5))

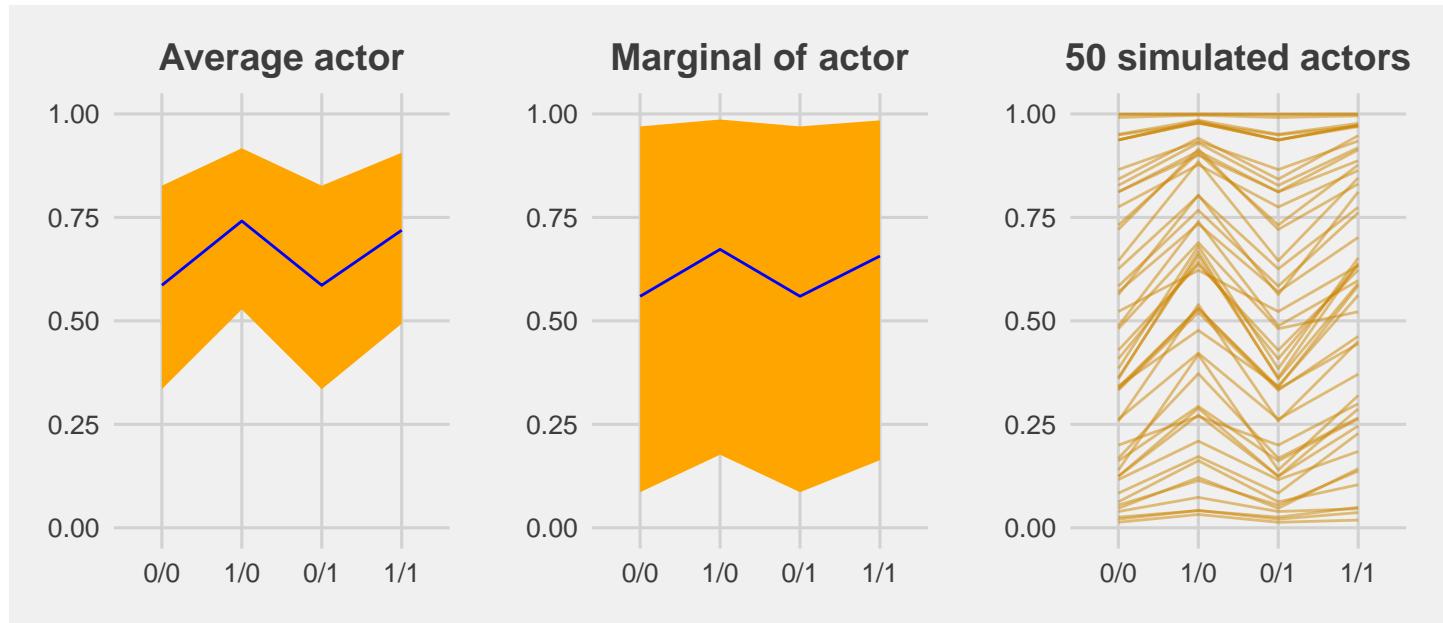
```

p3



For the finale, we'll stitch the three plots together.

```
library(gridExtra)
grid.arrange(p1, p2, p3, ncol = 3)
```



12.4.2.1 Bonus: Let's use `fitted()` this time.

We just made those plots using various wrangled versions of `post`, the data frame returned by `posterior_samples(b.12.4)`. If you followed along closely, part of what made that a great exercise is that it forced you to consider what the various vectors in `post` meant with respect to the model formula. But it's also handy to see how to do that from a different perspective. So in this section, we'll repeat that process by relying on the `fitted()` function, instead. We'll go in the same order, starting with the average actor.

```

nd <-
  tibble(prosoc_left = c(0, 1, 0, 1),
         condition   = c(0, 0, 1, 1))

(
  f <-
  fitted(b12.4,
    newdata = nd,
    re_formula = NA,
    probs = c(.1, .9)) %>%
  as_tibble() %>%
  bind_cols(nd) %>%
  mutate(condition = str_c(prosoc_left, "/", condition) %>%
    factor(. , levels = c("0/0", "1/0", "0/1", "1/1")))
)

```

```

## # A tibble: 4 x 6
##   Estimate Est.Error   Q10   Q90 prosoc_left condition
##     <dbl>     <dbl> <dbl> <dbl>     <dbl> <fct>
## 1     0.586     0.187  0.335  0.827      0 0/0
## 2     0.741     0.159  0.528  0.917      1 1/0
## 3     0.586     0.187  0.335  0.827      0 0/1
## 4     0.719     0.165  0.493  0.906      1 1/1

```

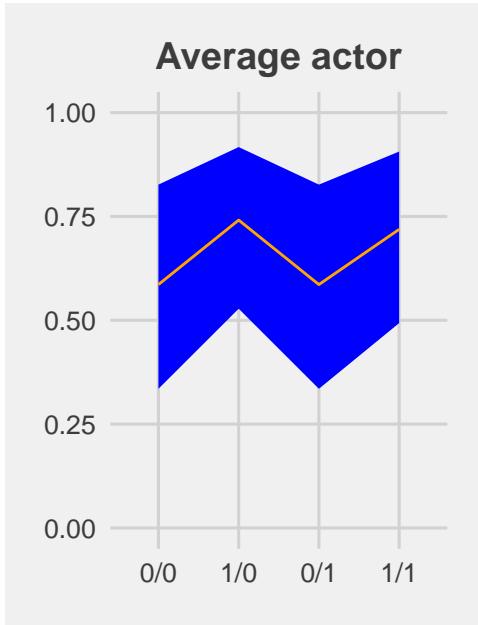
You should notice a few things. Since `b12.4` is a multilevel model, it had three predictors: `prosoc_left`, `condition`, and `actor`. However, our `nd` data only included the first two of those predictors. The reason `fitted()` permitted that was because we set `re_formula = NA`. When you do that, you tell `fitted()` to ignore group-level effects (i.e., focus only on the fixed effects). This was our `fitted()` version of ignoring the `r_` vectors returned by `posterior_samples()`. Here's the plot.

```

p4 <-
f %>%
ggplot(aes(x = condition, y = Estimate, group = 1)) +
  geom_ribbon(aes(ymin = Q10, ymax = Q90), fill = "blue") +
  geom_line(color = "orange1") +
  ggtitle("Average actor") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 14, hjust = .5))

```

p4



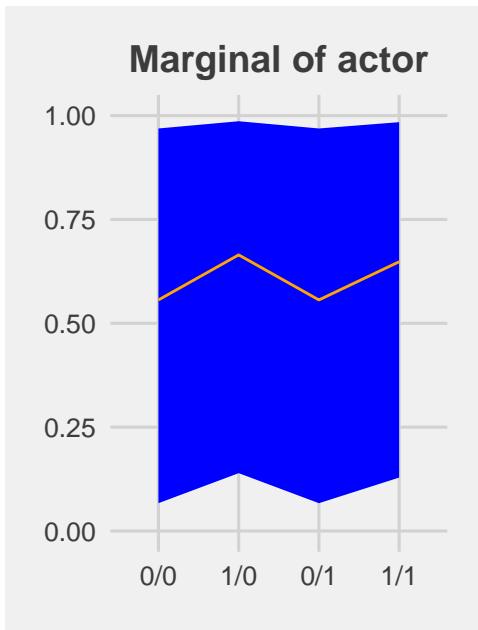
For marginal of actor, we can continue using the same `nd` data. This time we'll be sticking with the default `re_formula` setting, which will accommodate the multilevel nature of the model. However, we'll also be adding `allow_new_levels = T` and `sample_new_levels = "gaussian"`. The former will allow us to marginalize across the specific actors in our data and the latter will instruct `fitted()` to use the multivariate normal distribution implied by the random effects. It'll make more sense why I say *multivariate* normal by the end of the next chapter. For now, just go with it.

```
(f <-
  fitted(b12.4,
    newdata = nd,
    probs = c(.1, .9),
    allow_new_levels = T,
    sample_new_levels = "gaussian") %>%
  as_tibble() %>%
  bind_cols(nd) %>%
  mutate(condition = str_c(prosoc_left, "/", condition) %>%
    factor(. , levels = c("0/0", "1/0", "0/1", "1/1")))
)
```

	Estimate	Est.Error	Q10	Q90	prosoc_left	condition
1	0.556	0.330	0.0669	0.969	0	0/0
2	0.665	0.312	0.139	0.986	1	1/0
3	0.556	0.330	0.0669	0.969	0	0/1
4	0.648	0.316	0.129	0.984	1	1/1

Here's our `fitted()`-based marginal of actor plot.

```
p5 <-
f %>%
ggplot(aes(x = condition, y = Estimate, group = 1)) +
  geom_ribbon(aes(ymin = Q10, ymax = Q90), fill = "blue") +
  geom_line(color = "orange1") +
  ggtitle("Marginal of actor") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 14, hjust = .5))
```



For the simulated actors plot, we'll just amend our process from the last one. This time we're setting `summary = F`, in order to keep the iteration-specific results, and setting `nsamples = n_sim`. `n_sim` is just a name for the number of actors we'd like to simulate (i.e., 50, as in the text).

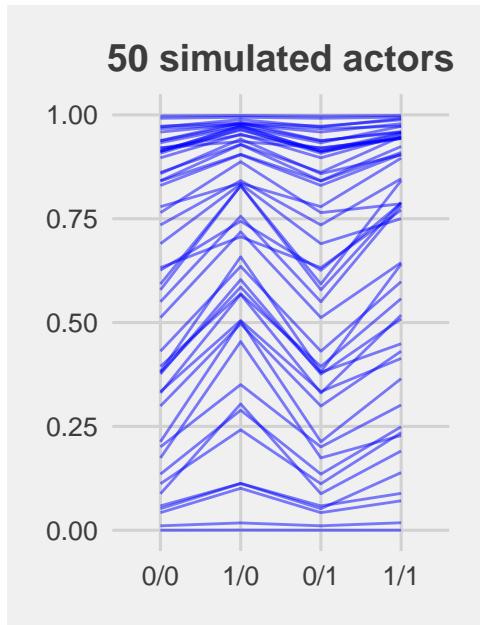
```
# how many simulated actors would you like?
n_sim <- 50

(
  f <-
  fitted(b12.4,
    newdata = nd,
    probs = c(.1, .9),
    allow_new_levels = T,
    sample_new_levels = "gaussian",
    summary = F,
    nsamples = n_sim) %>%
  as_tibble() %>%
  mutate(iter = 1:n_sim) %>%
  gather(key, value, -iter) %>%
  bind_cols(nd %>%
    transmute(condition = str_c(prosoc_left, "/", condition) %>%
      factor(., levels = c("0/0", "1/0", "0/1", "1/1")))) %>%
    expand(condition, iter = 1:n_sim)
  )

## # A tibble: 200 x 5
##       iter key   value condition iter1
##   <int> <chr> <dbl> <fct>     <int>
## 1     1 V1    0.330 0/0         1
## 2     2 V1    0.299 0/0         2
## 3     3 V1    0.841 0/0         3
## 4     4 V1    0.735 0/0         4
## 5     5 V1    0.858 0/0         5
## 6     6 V1    0.382 0/0         6
## 7     7 V1    0.690 0/0         7
## 8     8 V1    0.512 0/0         8
## 9     9 V1    0.912 0/0         9
## 10    10 V1   0.394 0/0        10
## # ... with 190 more rows
```

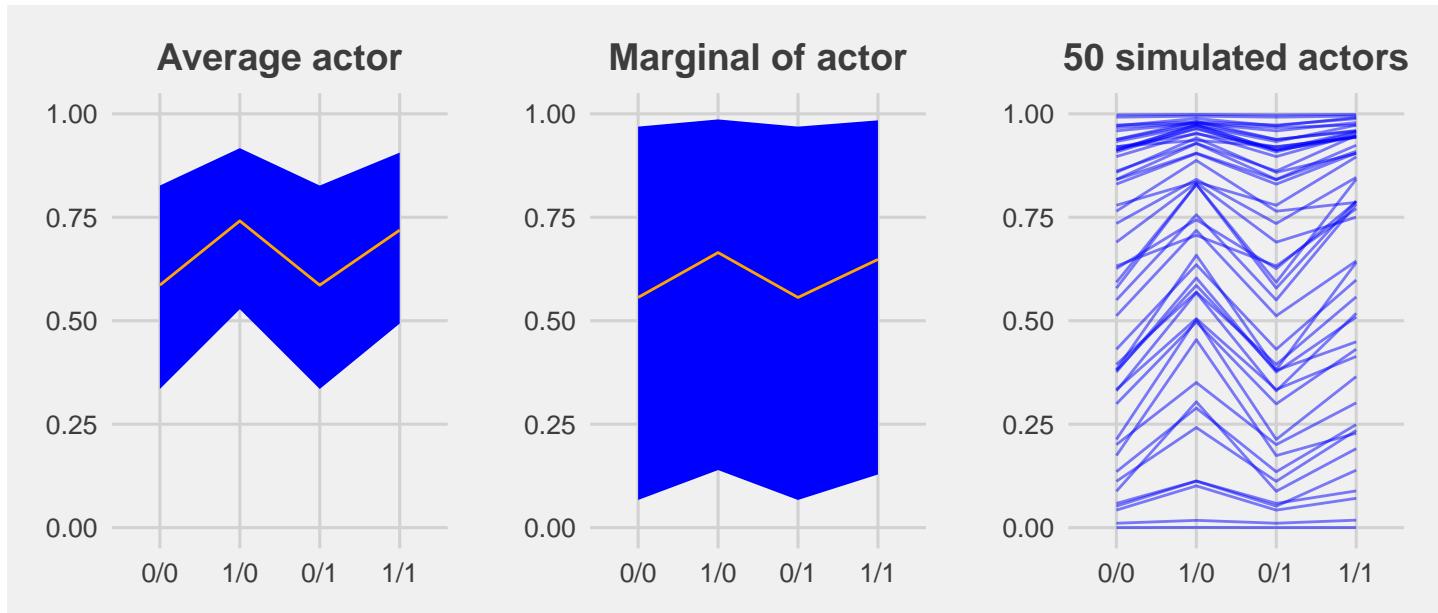
```
p6 <-
f %>%
  ggplot(aes(x = condition, y = value, group = iter)) +
  geom_line(alpha = 1/2, color = "blue") +
  ggttitle("50 simulated actors") +
  coord_cartesian(ylim = 0:1) +
  theme_fivethirtyeight() +
  theme(plot.title = element_text(size = 14, hjust = .5))
```

p6



Here they are altogether.

```
grid.arrange(p4, p5, p6, ncol = 3)
```



12.4.3 Focus and multilevel prediction.

First, let's load the `Kline` data.

```
# prep data
library(rethinking)
data(Kline)
d <- Kline
```

Switch out the packages, once again.

```
detach(package:rethinking, unload = T)
library(brms)
rm(Kline)
```

The statistical formula for our multilevel count model is

$$\begin{aligned} \text{total_tools}_i &\sim \text{Poisson}(\mu_i) \\ \log(\mu_i) &= \alpha + \alpha_{\text{culture}_i} + \beta \log(\text{population}_i) \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \\ \alpha_{\text{culture}} &\sim \text{Normal}(0, \sigma_{\text{culture}}) \\ \sigma_{\text{culture}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

With `brms`, we don't actually need to make the `logpop` or `society` variables. We're ready to fit the multilevel `Kline` model with the data in hand.

```
b12.6 <-
  brm(data = d, family = poisson,
       total_tools ~ 0 + intercept + log(population) +
       (1 | culture),
       prior = c(prior(normal(0, 10), class = b, coef = intercept),
                 prior(normal(0, 1), class = b),
                 prior(cauchy(0, 1), class = sd)),
       iter = 4000, warmup = 1000, cores = 3, chains = 3,
       seed = 12)
```

Note how we used the special `0 + intercept` syntax rather than using the default Intercept. This is because our predictor variable was not mean centered. For more info, see [here](#). Though we used the `0 + intercept` syntax for the fixed effect, it was not necessary for the random effect. Both ways work.

Here is the data-processing work for our variant of Figure 12.6.

```
nd <-
  tibble(population = seq(from = 1000, to = 400000, by = 5000),
        # to "simulate counterfactual societies, using the hyper-parameters" (p. 383),
        # we'll plug a new island into the `culture` variable
        culture      = "my_island")

p <-
  predict(b12.6,
        # this allows us to simulate values for our counterfactual island, "my_island"
        allow_new_levels = T,
        # here we explicitly tell brms we want to include the group-level effects
        re_formula = ~ (1 | culture),
```

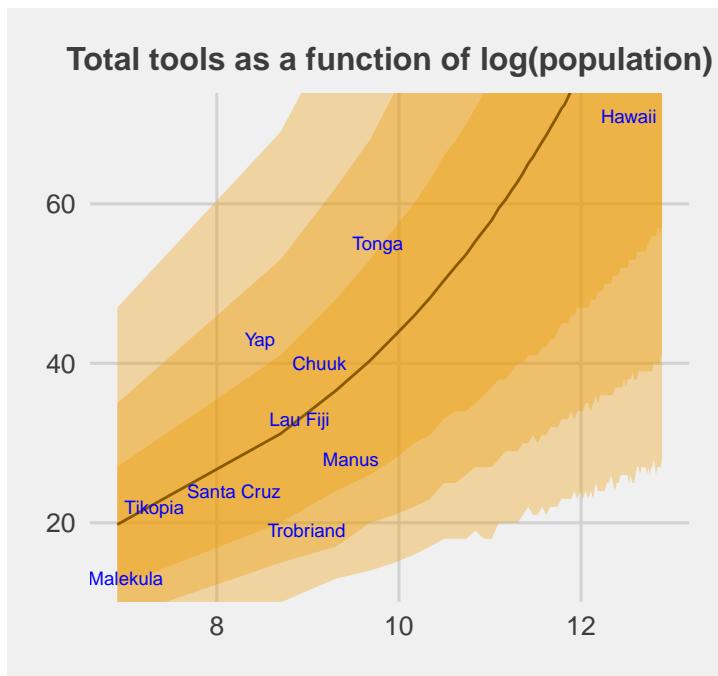
```
# from the brms manual, this uses the "(multivariate) normal distribution implied by
# the group-level standard deviations and correlations", which appears to be
# what McElreath did in the text.
sample_new_levels = "gaussian",
newdata = nd,
probs = c(.015, .055, .165, .835, .945, .985)) %>%
as_tibble() %>%
bind_cols(nd)

p %>%
glimpse()
```

```
## Observations: 80
## Variables: 10
## $ Estimate    <dbl> 19.78322, 31.16189, 36.55989, 40.31711, 43.44322, 45.94956, 48.21122, 50.344...
## $ Est.Error   <dbl> 9.864749, 14.103522, 16.554112, 18.521093, 20.545470, 22.052733, 23.445121, ...
## $ Q1.5       <dbl> 5.000, 10.000, 13.000, 14.000, 15.000, 16.000, 17.000, 17.985, 18.000, 18.00...
## $ Q5.5       <dbl> 8, 15, 17, 20, 21, 22, 23, 25, 25, 26, 27, 27, 27, 28, 29, 29, 29, 30, 30, 3...
## $ Q16.5      <dbl> 12, 20, 24, 26, 28, 30, 31, 33, 34, 34, 35, 36, 37, 38, 38, 39, 40, 40, 41, ...
## $ Q83.5      <dbl> 27, 41, 48, 53, 57, 60, 63, 66, 68, 70, 72, 74, 76, 78, 79, 81, 82, 83, 85, ...
## $ Q94.5      <dbl> 35.000, 53.000, 62.000, 68.000, 74.000, 78.000, 81.000, 85.000, 89.000, 91.0...
## $ Q98.5      <dbl> 47.000, 69.000, 82.000, 91.000, 98.015, 106.000, 109.000, 115.000, 120.015, ...
## $ population <dbl> 1000, 6000, 11000, 16000, 21000, 26000, 31000, 36000, 41000, 46000, 51000, 5...
## $ culture    <chr> "my_island", "my_island", "my_island", "my_island", "my_island", "my_island"...
```

For a detailed discussion on this way of using `brms::predict()`, see [Andrew MacDonald's great blogpost on this very figure](#). Here's what we've been working for:

```
p %>%
ggplot(aes(x = log(population), y = Estimate)) +
geom_ribbon(aes(ymin = Q1.5, ymax = Q98.5), fill = "orange2", alpha = 1/3) +
geom_ribbon(aes(ymin = Q5.5, ymax = Q94.5), fill = "orange2", alpha = 1/3) +
geom_ribbon(aes(ymin = Q16.5, ymax = Q83.5), fill = "orange2", alpha = 1/3) +
geom_line(color = "orange4") +
geom_text(data = d, aes(y = total_tools, label = culture),
size = 2.33, color = "blue") +
ggttitle("Total tools as a function of log(population)") +
coord_cartesian(ylim = range(d$total_tools)) +
theme_fivethirtyeight() +
theme(plot.title = element_text(size = 12, hjust = .5))
```



Glorious.

The envelope of predictions is a lot wider here than it was back in Chapter 10. This is a consequence of the varying intercepts, combined with the fact that there is much more variation in the data than a pure-Poisson model anticipates. (p. 384)

12.5 Summary Bonus: `tidybayes::spread_draws()`

A big part of this chapter, both what McElreath focused on in the text and even our plotting digression a bit above, focused on how to combine the fixed effects of a multilevel with the group-level. Given some binomial variable, criterion, and some group term, grouping variable, we've learned the simple multilevel model follows a form like

$$\begin{aligned} \text{criterion}_i &\sim \text{Binomial}(n_i \geq 1, p_i) \\ \text{logit}(p_i) &= \alpha + \alpha_{\text{grouping variable}_i} \\ \alpha &\sim \text{Normal}(0, 1) \\ \alpha_{\text{grouping variable}} &\sim \text{Normal}(0, \sigma_{\text{grouping variable}}) \\ \sigma_{\text{grouping variable}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

and we've been grappling with the relation between the grand mean α and the group-level deviations $\alpha_{\text{grouping variable}}$. For situations where we have the `brms::brm()` model fit in hand, we've been playing with various ways to use the iterations, particularly with either the `posterior_samples()` method and the `fitted()/predict()` method. Both are great. But (a) we have other options, which I'd like to share, and (b) if you're like me, you probably need more practice than following along with the examples in the text. In this bonus section, we are going to introduce two simplified models and then practice working with combining the grand mean various combinations of the random effects.

For our first step, we'll introduce the models.

12.5.1 Intercepts-only models with one or two grouping variables

If you recall, b12.4 was our first multilevel model with the chimps data. We can retrieve the model formula like so.

```
b12.4$formula
```

```
## pulled_left | trials(1) ~ 1 + prosoc_left + prosoc_left:condition + (1 | actor)
```

In addition to the model intercept and random effects for the individual chimps (i.e., `actor`), we also included fixed effects for the study conditions. For our bonus section, it'll be easier if we reduce this to a simple intercepts-only model with the sole `actor` grouping factor. That model will follow the form

$$\begin{aligned} \text{pulled_left}_i &\sim \text{Binomial}(n_i = 1, p_i) \\ \text{logit}(p_i) &= \alpha + \alpha_{\text{actor}_i} \\ \alpha &\sim \text{Normal}(0, 10) \\ \alpha_{\text{actor}} &\sim \text{Normal}(0, \sigma_{\text{actor}}) \\ \sigma_{\text{actor}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

Before we fit the model, you might recall that (a) we've already removed the `chimpanzees` data after saving the data as `d` and (b) we subsequently reassigned the `Kline` data to `d`. Instead of reloading the `rethinking` package to retrieve the `chimpanzees` data, we might also acknowledge that the data has also been saved within our `b12.4` fit object. [It's easy to forget such things.]

```
b12.4$data %>%
  glimpse()
```

```
## Observations: 504
## Variables: 4
## $ pulled_left <int> 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, ...
## $ prosoc_left <int> 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, ...
## $ condition <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ actor      <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

So there's no need to reload anything. Everything we need is already at hand. Let's fit the intercepts-only model.

```
b12.7 <-
  brm(data = b12.4$data, family = binomial,
       pulled_left ~ 1 + (1 | actor),
       prior = c(prior(normal(0, 10), class = Intercept),
                  prior(cauchy(0, 1), class = sd)),
       iter = 5000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = 0.95),
       seed = 12)
```

Here's the model summary:

```
print(b12.7)

## Family: binomial
##  Links: mu = logit
## Formula: pulled_left ~ 1 + (1 | actor)
##   Data: b12.4$data (Number of observations: 504)
## Samples: 4 chains, each with iter = 5000; warmup = 1000; thin = 1;
##          total post-warmup samples = 16000
##
## Group-Level Effects:
##   ~actor (Number of levels: 7)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##   sd(Intercept)    2.20      0.89     1.10     4.55        2859 1.00
## 
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##   Intercept      0.76      0.92    -0.93     2.70        2370 1.00
## 
```

```
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Now recall that our competing cross-classified model, b12.5 added random effects for the trial blocks. Here was that formula.

```
b12.5$formula
```

```
## pulled_left | trials(1) ~ prosoc_left + (1 | actor) + (1 | block) + prosoc_left:condition
```

And, of course, we can retrieve the data from that model, too.

```
b12.5$data %>%
  glimpse()
```

```
## Observations: 504
## Variables: 5
## $ pulled_left <int> 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, ...
## $ prosoc_left <int> 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, ...
## $ condition <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ actor      <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ block       <int> 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, ...
```

It's the same data we used from the b12.4 model, but with the addition of the block index. With those data in hand, we can fit the intercepts-only version of our cross-classified model. This model formula follows the form

$$\begin{aligned} \text{pulled_left}_i &\sim \text{Binomial}(n_i = 1, p_i) \\ \text{logit}(p_i) &= \alpha + \alpha_{\text{actor}_i} + \alpha_{\text{block}_i} \\ \alpha &\sim \text{Normal}(0, 10) \\ \alpha_{\text{actor}} &\sim \text{Normal}(0, \sigma_{\text{actor}}) \\ \alpha_{\text{block}} &\sim \text{Normal}(0, \sigma_{\text{block}}) \\ \sigma_{\text{actor}} &\sim \text{HalfCauchy}(0, 1) \\ \sigma_{\text{block}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

Fit the model.

```
b12.8 <-
  brm(data = b12.5$data, family = binomial,
    pulled_left | trials(1) ~ 1 + (1 | actor) + (1 | block),
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(cauchy(0, 1), class = sd)),
    iter = 5000, warmup = 1000, chains = 4, cores = 4,
    control = list(adapt_delta = 0.95),
    seed = 12)
```

Here's the summary.

```
print(b12.8)

## Family: binomial
## Links: mu = logit
## Formula: pulled_left | trials(1) ~ 1 + (1 | actor) + (1 | block)
## Data: b12.5$data (Number of observations: 504)
## Samples: 4 chains, each with iter = 5000; warmup = 1000; thin = 1;
```

```

##      total post-warmup samples = 16000
##
## Group-Level Effects:
## ~actor (Number of levels: 7)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)    2.24     0.94    1.09    4.53      4049 1.00
##
## ~block (Number of levels: 6)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)    0.22     0.18    0.01    0.67      6647 1.00
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## Intercept      0.82     0.94   -0.92    2.79      3460 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Now we've fit our two intercepts-only models, let's get to the heart of this section. We are going to practice four methods for working with the posterior samples. Each method will revolve around a different primary function. In order, they are

- `brms::posterior_samples()`
- `brms::coef()`
- `brms::fitted()`
- `tidybayes::spread_draws()`

We've already had some practice with the first three, but I hope this section will make them even more clear. The `tidybayes::spread_draws()` method will be new, to us. I think you'll find it's a handy alternative.

With each of the four methods, we'll practice three different model summaries.

- Getting the posterior draws for the `actor`-level estimates from the `b12.7` model
- Getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, averaging over the levels of `block`
- Getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, based on `block == 1`

So to be clear, our goal is to accomplish those three tasks with four methods, each of which should yield equivalent results.

12.5.2 `brms::posterior_samples()`

To warm up, let's take a look at the structure of the `posterior_samples()` output for the simple `b12.7` model.

```
posterior_samples(b12.7) %>% str()
```

```

## 'data.frame': 16000 obs. of 10 variables:
## $ b_Intercept : num 1.53 1.3 1.85 1.97 1.85 ...
## $ sd_actor__Intercept : num 1.76 1.34 1.8 1.6 1.98 ...
## $ r_actor[1,Intercept]: num -2.08 -1.35 -2.27 -2.24 -2.22 ...
## $ r_actor[2,Intercept]: num 2.44 2.19 3.17 1.61 2.24 ...
## $ r_actor[3,Intercept]: num -2.04 -1.85 -2.38 -2.08 -3.03 ...
## $ r_actor[4,Intercept]: num -2.62 -1.53 -2.16 -2.72 -2.26 ...
## $ r_actor[5,Intercept]: num -1.56 -2.09 -2.32 -2.22 -2.33 ...
## $ r_actor[6,Intercept]: num -1.046 -0.873 -0.829 -1.539 -1.272 ...
## $ r_actor[7,Intercept]: num 0.947 0.273 -0.241 0.574 0.785 ...
## $ lp_-- : num -283 -286 -283 -285 -283 ...

```

The `b_Intercept` vector corresponds to the α term in the statistical model. The second vector, `sd_actor_Intercept`, corresponds to the σ_{actor} term. And the next 7 vectors beginning with the `r_actor` suffix are the α_{actor} deviations from the grand mean, α . Thus if we wanted to get the model-implied probability for our first chimp, we'd add `b_Intercept` to `r_actor[1,Intercept]` and then take the inverse logit.

```
posterior_samples(b12.7) %>%
  transmute(`chimp 1's average probability of pulling left` = (b_Intercept + `r_actor[1,Intercept]`) %>% inv_logit()
head()

##   chimp 1's average probability of pulling left
## 1                               0.3666511
## 2                               0.4891179
## 3                               0.3965780
## 4                               0.4309511
## 5                               0.4097169
## 6                               0.4485629
```

To complete our first task, then, of getting the posterior draws for the actor-level estimates from the `b12.7` model, we can do that in bulk.

```
p1 <-
  posterior_samples(b12.7) %>%
  transmute(`chimp 1's average probability of pulling left` = b_Intercept + `r_actor[1,Intercept]`,
            `chimp 2's average probability of pulling left` = b_Intercept + `r_actor[2,Intercept]`,
            `chimp 3's average probability of pulling left` = b_Intercept + `r_actor[3,Intercept]`,
            `chimp 4's average probability of pulling left` = b_Intercept + `r_actor[4,Intercept]`,
            `chimp 5's average probability of pulling left` = b_Intercept + `r_actor[5,Intercept]`,
            `chimp 6's average probability of pulling left` = b_Intercept + `r_actor[6,Intercept]`,
            `chimp 7's average probability of pulling left` = b_Intercept + `r_actor[7,Intercept]`) %>%
  mutate_all(inv_logit_scaled)

str(p1)

## #> #> 'data.frame': 16000 obs. of 7 variables:
## #> $ chimp 1's average probability of pulling left: num 0.367 0.489 0.397 0.431 0.41 ...
## #> $ chimp 2's average probability of pulling left: num 0.982 0.971 0.993 0.973 0.984 ...
## #> $ chimp 3's average probability of pulling left: num 0.375 0.367 0.37 0.472 0.236 ...
## #> $ chimp 4's average probability of pulling left: num 0.252 0.443 0.424 0.319 0.399 ...
## #> $ chimp 5's average probability of pulling left: num 0.493 0.313 0.386 0.436 0.383 ...
## #> $ chimp 6's average probability of pulling left: num 0.619 0.606 0.735 0.605 0.642 ...
## #> $ chimp 7's average probability of pulling left: num 0.922 0.829 0.833 0.927 0.933 ...
```

One of the things I really like about this method is the `b_Intercept + r_actor[i,Intercept]` part of the code makes it very clear, to me, how the `posterior_samples()` columns correspond to the statistical model, $\text{logit}(p_i) = \alpha + \alpha_{\text{actor}_i}$. This method easily extends to our next task, getting the posterior draws for the actor-level estimates from the cross-classified `b12.8` model, averaging over the levels of `block`. In fact, other than switching out `b12.7` for `b12.8`, the method is identical.

```
p2 <-
  posterior_samples(b12.8) %>%
  transmute(`chimp 1's average probability of pulling left` = b_Intercept + `r_actor[1,Intercept]`,
            `chimp 2's average probability of pulling left` = b_Intercept + `r_actor[2,Intercept]`,
            `chimp 3's average probability of pulling left` = b_Intercept + `r_actor[3,Intercept]`,
            `chimp 4's average probability of pulling left` = b_Intercept + `r_actor[4,Intercept]`,
            `chimp 5's average probability of pulling left` = b_Intercept + `r_actor[5,Intercept]`,
            `chimp 6's average probability of pulling left` = b_Intercept + `r_actor[6,Intercept]`,
            `chimp 7's average probability of pulling left` = b_Intercept + `r_actor[7,Intercept]`) %>%
  mutate_all(inv_logit_scaled)

str(p2)
```

```
## 'data.frame': 16000 obs. of 7 variables:
## $ chimp 1's average probability of pulling left: num 0.403 0.437 0.458 0.421 0.416 ...
## $ chimp 2's average probability of pulling left: num 0.993 0.99 0.991 1 0.999 ...
## $ chimp 3's average probability of pulling left: num 0.406 0.419 0.385 0.343 0.384 ...
## $ chimp 4's average probability of pulling left: num 0.353 0.347 0.31 0.286 0.429 ...
## $ chimp 5's average probability of pulling left: num 0.391 0.388 0.352 0.478 0.36 ...
## $ chimp 6's average probability of pulling left: num 0.672 0.683 0.646 0.634 0.633 ...
## $ chimp 7's average probability of pulling left: num 0.904 0.896 0.889 0.927 0.897 ...
```

The reason we can still get away with this is because the grand mean in the `b12.8` model is the grand mean across all levels of `actor` and `block`. AND it's the case that the `r_actor` and `r_block` vectors returned by `posterior_samples(b12.8)` are all in deviation metrics—execute `posterior_samples(b12.8) %>% glimpse()` if it will help you follow along. So if we simply leave out the `r_block` vectors, we are ignoring the specific `block`-level deviations, effectively averaging over them.

Now for our third task, we've decided we wanted to retrieve the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, based on `block == 1`. To get the chimp-specific estimates for the first `block`, we simply add `+ r_block[1, Intercept]` to the end of each formula.

```
p3 <-
```

```
posterior_samples(b12.8) %>%
  transmute(`chimp 1's average probability of pulling left` = b_Intercept + `r_actor[1,Intercept]` + `r_block[1,Intercept]`,
           `chimp 2's average probability of pulling left` = b_Intercept + `r_actor[2,Intercept]` + `r_block[2,Intercept]`,
           `chimp 3's average probability of pulling left` = b_Intercept + `r_actor[3,Intercept]` + `r_block[3,Intercept]`,
           `chimp 4's average probability of pulling left` = b_Intercept + `r_actor[4,Intercept]` + `r_block[4,Intercept]`,
           `chimp 5's average probability of pulling left` = b_Intercept + `r_actor[5,Intercept]` + `r_block[5,Intercept]`,
           `chimp 6's average probability of pulling left` = b_Intercept + `r_actor[6,Intercept]` + `r_block[6,Intercept]`,
           `chimp 7's average probability of pulling left` = b_Intercept + `r_actor[7,Intercept]` + `r_block[7,Intercept]`)
  mutate_all(inv_logit_scaled)
```

```
str(p3)
```

```
## 'data.frame': 16000 obs. of 7 variables:
## $ chimp 1's average probability of pulling left: num 0.278 0.468 0.457 0.422 0.408 ...
## $ chimp 2's average probability of pulling left: num 0.987 0.991 0.991 1 0.999 ...
## $ chimp 3's average probability of pulling left: num 0.28 0.45 0.384 0.343 0.376 ...
## $ chimp 4's average probability of pulling left: num 0.237 0.376 0.309 0.287 0.421 ...
## $ chimp 5's average probability of pulling left: num 0.268 0.417 0.351 0.479 0.353 ...
## $ chimp 6's average probability of pulling left: num 0.538 0.709 0.646 0.635 0.625 ...
## $ chimp 7's average probability of pulling left: num 0.843 0.907 0.889 0.927 0.894 ...
```

Again, I like this method because of how close the wrangling code within `transmute()` is to the statistical model formula. I wrote a lot of code like this in my early days of working with these kinds of models, and I think the pedagogical insights were helpful. But this method has its limitations. It works fine if you're working with some small number of groups. But that's a lot of repetitious code and it would be utterly un-scalable to situations where you have 50 or 500 levels in your grouping variable. We need alternatives.

12.5.3 `brms::coef()`

First, let's review what the `coef()` function returns.

```
coef(b12.7)
```

```
## $actor
## , , Intercept
##
##   Estimate Est.Error    Q2.5    Q97.5
## 1 -0.3228553 0.2401400 -0.7960922 0.1402474
## 2  4.8485079 1.5383234  2.8423349 8.6052735
```

```
## 3 -0.6191537 0.2513396 -1.1214545 -0.1412591
## 4 -0.6156347 0.2458075 -1.1090964 -0.1409848
## 5 -0.3224140 0.2362941 -0.7925155 0.1336236
## 6 0.5808851 0.2480937 0.1046163 1.0769954
## 7 2.0847013 0.3749189 1.4048612 2.8775507
```

By default, we get the familiar summaries for mean performances for each of our seven chimps. These, of course, are in the log-odds metric and simply tacking on `inv_logit_scaled()` isn't going to fully get the job done. So to get things in the probability metric, we'll want to first set `summary = F` in order to work directly with un-summarized samples and then wrangle quite a bit. Part of the wrangling challenge is because `coef()` returns a list, rather than a data frame. With that in mind, the code for our first task of getting the posterior draws for the `actor`-level estimates from the `b12.7` model looks like so.

```
c1 <-
  coef(b12.7, summary = F)$actor[, , ] %>%
  as_tibble() %>%
  gather() %>%
  mutate(key    = str_c("chimp ", key, "'s average probability of pulling left"),
         value = inv_logit_scaled(value),
         # we need an iteration index for `spread()` to work properly
         iter   = rep(1:16000, times = 7)) %>%
  spread(key = key, value = value)
```

```
str(c1)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   $ iter           : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.367 0.489 0.397 0.431 0.41 ...
##   $ chimp 2's average probability of pulling left: num 0.982 0.971 0.993 0.973 0.984 ...
##   $ chimp 3's average probability of pulling left: num 0.375 0.367 0.37 0.472 0.236 ...
##   $ chimp 4's average probability of pulling left: num 0.252 0.443 0.424 0.319 0.399 ...
##   $ chimp 5's average probability of pulling left: num 0.493 0.313 0.386 0.436 0.383 ...
##   $ chimp 6's average probability of pulling left: num 0.619 0.606 0.735 0.605 0.642 ...
##   $ chimp 7's average probability of pulling left: num 0.922 0.829 0.833 0.927 0.933 ...
```

So with this method, you get a little practice with three-dimensional indexing, which is a good skill to have. Now let's extend it to our second task, getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, averaging over the levels of `block`.

```
c2 <-
  coef(b12.8, summary = F)$actor[, , ] %>%
  as_tibble() %>%
  gather() %>%
  mutate(key    = str_c("chimp ", key, "'s average probability of pulling left"),
         value = inv_logit_scaled(value),
         iter   = rep(1:16000, times = 7)) %>%
  spread(key = key, value = value)
```

```
str(c2)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   $ iter           : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.403 0.437 0.458 0.421 0.416 ...
##   $ chimp 2's average probability of pulling left: num 0.993 0.99 0.991 1 0.999 ...
##   $ chimp 3's average probability of pulling left: num 0.406 0.419 0.385 0.343 0.384 ...
##   $ chimp 4's average probability of pulling left: num 0.353 0.347 0.31 0.286 0.429 ...
##   $ chimp 5's average probability of pulling left: num 0.391 0.388 0.352 0.478 0.36 ...
##   $ chimp 6's average probability of pulling left: num 0.672 0.683 0.646 0.634 0.633 ...
##   $ chimp 7's average probability of pulling left: num 0.904 0.896 0.889 0.927 0.897 ...
```

As with our `posterior_samples()` method, this code was near identical to the block, above. All we did was switch out `b12.7` for `b12.8`. [Okay, we removed a line of annotations. But that doesn't really count.] We should point something out, though. Consider what `coef()` yields when working with a cross-classified model.

```
coef(b12.8)
```

```
## $actor
## , , Intercept
##
##      Estimate Est.Error      Q2.5      Q97.5
## 1 -0.3309382 0.2637471 -0.85912023 0.1813434
## 2  4.9125023 1.6005794  2.83104573 8.9988482
## 3 -0.6190318 0.2758194 -1.16390465 -0.0835298
## 4 -0.6236739 0.2698544 -1.15833584 -0.1057192
## 5 -0.3275376 0.2679562 -0.86405363 0.1905555
## 6  0.5847788 0.2735922  0.05012094 1.1297621
## 7  2.0884696 0.3942676  1.35642012 2.9075374
##
## 
## $block
## , , Intercept
##
##      Estimate Est.Error      Q2.5      Q97.5
## 1  0.6359211 0.9506609 -1.1469973 2.625571
## 2  0.8724930 0.9394898 -0.9004279 2.823420
## 3  0.8727196 0.9402097 -0.8879903 2.844786
## 4  0.8000079 0.9398187 -0.9554111 2.784783
## 5  0.8016466 0.9403898 -0.9509095 2.759832
## 6  0.9214440 0.9414923 -0.8313673 2.896529
```

Now we have a list of two elements, one for `actor` and one for `block`. What might not be immediately obvious is that the summaries returned by one grouping level are based off of averaging over the other. Although this made our second task easy, it provides a challenge for our third task, getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, based on `block == 1`. To accomplish that, we'll need to bring in `ranef()`. Let's review what that returns.

```
ranef(b12.8)
```

```
## $actor
## , , Intercept
##
##      Estimate Est.Error      Q2.5      Q97.5
## 1 -1.1479441 0.9548717 -3.155849 0.6391628
## 2  4.0954964 1.6607249  1.735123 8.1131626
## 3 -1.4360377 0.9537786 -3.446306 0.3775506
## 4 -1.4406798 0.9513512 -3.440700 0.3242996
## 5 -1.1445435 0.9530794 -3.182475 0.6500693
## 6 -0.2322271 0.9511768 -2.225866 1.5415640
## 7  1.2714637 0.9818530 -0.719958 3.1740264
##
## 
## $block
## , , Intercept
##
##      Estimate Est.Error      Q2.5      Q97.5
## 1 -0.18108484 0.2278906 -0.7372273 0.1270714
## 2  0.05548710 0.1866952 -0.2827610 0.4917787
## 3  0.05571371 0.1851168 -0.2820499 0.4892504
## 4 -0.01699796 0.1824383 -0.4142501 0.3615457
## 5 -0.01535930 0.1826817 -0.4145747 0.3584713
## 6  0.10443814 0.2005518 -0.2168373 0.5870416
```

The format of the `ranef()` output is identical to that from `coef()`. However, the summaries are in the deviance metric. They're all centered around zero, which corresponds to the part of the statistical model that specifies how $\alpha_{\text{block}} \sim \text{Normal}(0, \sigma_{\text{block}})$. So then, if we want to continue using our `coef()` method, we'll need to augment it with `ranef()` to accomplish our last task.

```
c3 <-
  coef(b12.8, summary = F)$actor[, , ] %>%
  as_tibble() %>%
  gather() %>%
  # here we add in the `block == 1` deviations from the grand mean
  mutate(value = value + ranef(b12.8, summary = F)$block[, 1, ] %>% rep(., times = 7)) %>%
  mutate(key    = str_c("chimp ", key, "'s average probability of pulling left"),
         value = inv_logit_scaled(value),
         iter   = rep(1:16000, times = 7)) %>%
  spread(key = key, value = value)

str(c3)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   $ iter           : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.278 0.468 0.457 0.422 0.408 ...
##   $ chimp 2's average probability of pulling left: num 0.987 0.991 0.991 1 0.999 ...
##   $ chimp 3's average probability of pulling left: num 0.28 0.45 0.384 0.343 0.376 ...
##   $ chimp 4's average probability of pulling left: num 0.237 0.376 0.309 0.287 0.421 ...
##   $ chimp 5's average probability of pulling left: num 0.268 0.417 0.351 0.479 0.353 ...
##   $ chimp 6's average probability of pulling left: num 0.538 0.709 0.646 0.635 0.625 ...
##   $ chimp 7's average probability of pulling left: num 0.843 0.907 0.889 0.927 0.894 ...
```

One of the nicest things about the `coef()` method is how it scales well. This code is no more burdensome for 5 group levels than it is for 5000. It's also a post-processing version of the distinction McElreath made on page 372 between the two equivalent ways you might define a Gaussian:

$$\text{Normal}(10, 1)$$

and

$$10 + \text{Normal}(0, 1)$$

Conversely, it can be a little abstract. Let's keep expanding our options.

12.5.4 brms::fitted()

As is often the case, we're going to want to define our predictor values for `fitted()`.

```
(nd <- b12.7$data %>% distinct(actor))
```

```
##   actor
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
## 6     6
## 7     7
```

Now we have our new data, `nd`, here's how we might use `fitted()` to accomplish our first task, getting the posterior draws for the `actor`-level estimates from the `b12.7` model.

```
f1 <-
  fitted(b12.7,
    newdata = nd,
    summary = F,
    # within `fitted()`, this line does the same work that
    # `inv_logit_scaled()` did with the other two methods
    scale = "response") %>%
  as_tibble() %>%
  set_names(str_c("chimp ", 1:7, "'s average probability of pulling left"))

str(f1)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 7 variables:
## $ chimp 1's average probability of pulling left: num 0.367 0.489 0.397 0.431 0.41 ...
## $ chimp 2's average probability of pulling left: num 0.982 0.971 0.993 0.973 0.984 ...
## $ chimp 3's average probability of pulling left: num 0.375 0.367 0.37 0.472 0.236 ...
## $ chimp 4's average probability of pulling left: num 0.252 0.443 0.424 0.319 0.399 ...
## $ chimp 5's average probability of pulling left: num 0.493 0.313 0.386 0.436 0.383 ...
## $ chimp 6's average probability of pulling left: num 0.619 0.606 0.735 0.605 0.642 ...
## $ chimp 7's average probability of pulling left: num 0.922 0.829 0.833 0.927 0.933 ...
```

This scales reasonably well. But might not work well if the vectors you wanted to rename didn't follow a serial order, like ours. If you're willing to pay with a few more lines of wrangling code, this method is more general, but still scalable.

```
f1 <-
  fitted(b12.7,
    newdata = nd,
    summary = F,
    scale = "response") %>%
  as_tibble() %>%
  # you'll need this line to make the `spread()` line work properly
  mutate(iter = 1:n()) %>%
  gather(key, value, -iter) %>%
  mutate(key = str_replace(key, "V", "chimp ")) %>%
  mutate(key = str_c(key, "'s average probability of pulling left")) %>%
  spread(key = key, value = value)

str(f1)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   : int 1 2 3 4 5 6 7 8 9 10 ...
## $ iter
## $ chimp 1's average probability of pulling left: num 0.367 0.489 0.397 0.431 0.41 ...
## $ chimp 2's average probability of pulling left: num 0.982 0.971 0.993 0.973 0.984 ...
## $ chimp 3's average probability of pulling left: num 0.375 0.367 0.37 0.472 0.236 ...
## $ chimp 4's average probability of pulling left: num 0.252 0.443 0.424 0.319 0.399 ...
## $ chimp 5's average probability of pulling left: num 0.493 0.313 0.386 0.436 0.383 ...
## $ chimp 6's average probability of pulling left: num 0.619 0.606 0.735 0.605 0.642 ...
## $ chimp 7's average probability of pulling left: num 0.922 0.829 0.833 0.927 0.933 ...
```

Now unlike with the previous two methods, our `fitted()` method will not allow us to simply switch out `b12.7` for `b12.8` to accomplish our second task of getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, averaging over the levels of `block`. This is because when we use `fitted()` in combination with its `newdata` argument, the function expects us to define values for all the predictor variables in the formula. Because the `b12.8` model has both `actor` and `block` grouping variables as predictors, the default requires we include both in our new data. But if we were to specify a value for `block` in the `nd` data, we would no longer be averaging over the levels of `block` anymore; we'd be selecting one of the levels of `block` in particular, which we don't yet want to do. Happily, `brms::fitted()` has a `re_formula` argument. If we would like to average out `block`, we simply drop it from the formula. Here's how to do so.

```
f2 <-
  fitted(b12.8,
    newdata = nd,
    # this line allows us to average over the levels of `block`
    re_formula = pulled_left ~ 1 + (1 | actor),
    summary = F,
    scale = "response") %>%
  as_tibble() %>%
  set_names(str_c("chimp ", 1:7, "'s average probability of pulling left"))

str(f2)

## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 7 variables:
## $ chimp 1's average probability of pulling left: num 0.403 0.437 0.458 0.421 0.416 ...
## $ chimp 2's average probability of pulling left: num 0.993 0.99 0.991 1 0.999 ...
## $ chimp 3's average probability of pulling left: num 0.406 0.419 0.385 0.343 0.384 ...
## $ chimp 4's average probability of pulling left: num 0.353 0.347 0.31 0.286 0.429 ...
## $ chimp 5's average probability of pulling left: num 0.391 0.388 0.352 0.478 0.36 ...
## $ chimp 6's average probability of pulling left: num 0.672 0.683 0.646 0.634 0.633 ...
## $ chimp 7's average probability of pulling left: num 0.904 0.896 0.889 0.927 0.897 ...
```

If we want to use `fitted()` for our third task of getting the posterior draws for the `actor`-level estimates from the cross-classified `b12.8` model, based on `block == 1`, we'll need to augment our `nd` data.

```
(

nd <-
b12.8$data %>%
distinct(actor) %>%
mutate(block = 1)
)

##   actor block
## 1     1     1
## 2     2     1
## 3     3     1
## 4     4     1
## 5     5     1
## 6     6     1
## 7     7     1
```

This time, we no longer need that `re_formula` argument.

```
f3 <-
  fitted(b12.8,
    newdata = nd,
    summary = F,
    scale = "response") %>%
  as_tibble() %>%
  set_names(str_c("chimp ", 1:7, "'s average probability of pulling left"))

str(f3)

## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 7 variables:
## $ chimp 1's average probability of pulling left: num 0.278 0.468 0.457 0.422 0.408 ...
## $ chimp 2's average probability of pulling left: num 0.987 0.991 0.991 1 0.999 ...
## $ chimp 3's average probability of pulling left: num 0.28 0.45 0.384 0.343 0.376 ...
## $ chimp 4's average probability of pulling left: num 0.237 0.376 0.309 0.287 0.421 ...
## $ chimp 5's average probability of pulling left: num 0.268 0.417 0.351 0.479 0.353 ...
## $ chimp 6's average probability of pulling left: num 0.538 0.709 0.646 0.635 0.625 ...
## $ chimp 7's average probability of pulling left: num 0.843 0.907 0.889 0.927 0.894 ...
```

Let's learn one more option.

12.5.5 `tidybayes::spread_draws()`

Up till this point, we've really only used the `tidybayes` package for plotting (e.g., with `geom_halfeyeh()`) and summarizing (e.g., with `median_qi()`). But `tidybayes` is more general; it offers a handful of convenience functions for wrangling posterior draws from a tidyverse perspective. One such function is `spread_draws()`, which you can learn all about in Matthew Kay's vignette [Extracting and visualizing tidy draws from brms models](#). Let's take a look at how we'll be using it.

```
library(tidybayes)

b12.7 %>%
  spread_draws(b_Intercept, r_actor[actor,])

## # A tibble: 112,000 x 6
## # Groups:   actor [7]
##   .chain .iteration .draw b_Intercept actor r_actor
##   <int>     <int> <int>     <dbl> <int>    <dbl>
## 1 1         1       1       1      1.53    1    -2.08
## 2 2         1       1       1      1.53    2     2.44
## 3 3         1       1       1      1.53    3    -2.04
## 4 4         1       1       1      1.53    4    -2.62
## 5 5         1       1       1      1.53    5    -1.56
## 6 6         1       1       1      1.53    6    -1.05
## 7 7         1       1       1      1.53    7    0.947
## 8 8         1       2       2      1.30    1    -1.35
## 9 9         1       2       2      1.30    2     2.19
## 10 10        1       2       2      1.30    3    -1.85
## # ... with 111,990 more rows
```

First, notice `tidybayes::spread_draws()` took the model fit itself, `b12.7`, as input. No need for `posterior_samples()`. Now, notice we fed it two additional arguments. By the first argument, we requested `spread_draws()` extract the posterior samples for the `b_Intercept`. By the second argument, `r_actor[actor,]`, we instructed `spread_draws()` to extract all the random effects for the `actor` variable. Also notice how within the brackets `[]` we specified `actor`, which then became the name of the column in the output that indexed the levels of the grouping variable `actor`. By default, the code returns the posterior samples for all the levels of `actor`. However, had we only wanted those from chimps #1 and #3, we might use typical tidyverse-style indexing. E.g.,

```
b12.7 %>%
  spread_draws(b_Intercept, r_actor[actor,]) %>%
  filter(actor %in% c(1, 3))
```

```
## # A tibble: 32,000 x 6
## # Groups:   actor [2]
##   .chain .iteration .draw b_Intercept actor r_actor
##   <int>     <int> <int>     <dbl> <int>    <dbl>
## 1 1         1       1       1      1.53    1    -2.08
## 2 2         1       1       1      1.53    3    -2.04
## 3 3         1       2       2      1.30    1    -1.35
## 4 4         1       2       2      1.30    3    -1.85
## 5 5         1       3       3      1.85    1    -2.27
## 6 6         1       3       3      1.85    3    -2.38
## 7 7         1       4       4      1.97    1    -2.24
## 8 8         1       4       4      1.97    3    -2.08
## 9 9         1       5       5      1.85    1    -2.22
## 10 10        1       5       5      1.85    3    -3.03
## # ... with 31,990 more rows
```

Also notice those first three columns. By default, `spread_draws()` extracted information about which Markov chain a given draw was from, which iteration a given draw was within a given chain, and which draw from an overall standpoint. If it helps to keep track of which vector indexed what, consider this.

```
b12.7 %>%
  spread_draws(b_Intercept, r_actor[actor,]) %>%
  ungroup() %>%
  select(.chain: .draw) %>%
  gather() %>%
  group_by(key) %>%
  summarise(min = min(value),
            max = max(value))
```

```
## # A tibble: 3 x 3
##   key      min   max
##   <chr>    <dbl> <dbl>
## 1 .chain      1     4
## 2 .draw       1 16000
## 3 .iteration  1    4000
```

Above we simply summarized each of the three variables by their minimum and maximum values. If you recall that we fit `b12.7` with four Markov chains, each with 4000 post-warmup iterations, hopefully it'll make sense what each of those three variables index.

Now we've done a little clarification, let's use `spread_draws()` to accomplish our first task, getting the posterior draws for the actor-level estimates from the `b12.7` model.

```
s1 <-
b12.7 %>%
  spread_draws(b_Intercept, r_actor[actor,]) %>%
  mutate(p = (b_Intercept + r_actor) %>% inv_logit_scaled()) %>%
  select(.draw, actor, p) %>%
  ungroup() %>%
  mutate(actor = str_c("chimp ", actor, "'s average probability of pulling left")) %>%
  spread(value = p, key = actor)

str(s1)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   $ .draw                  : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.367 0.489 0.397 0.431 0.41 ...
##   $ chimp 2's average probability of pulling left: num 0.982 0.971 0.993 0.973 0.984 ...
##   $ chimp 3's average probability of pulling left: num 0.375 0.367 0.37 0.472 0.236 ...
##   $ chimp 4's average probability of pulling left: num 0.252 0.443 0.424 0.319 0.399 ...
##   $ chimp 5's average probability of pulling left: num 0.493 0.313 0.386 0.436 0.383 ...
##   $ chimp 6's average probability of pulling left: num 0.619 0.606 0.735 0.605 0.642 ...
##   $ chimp 7's average probability of pulling left: num 0.922 0.829 0.833 0.927 0.933 ...
```

The method remains essentially the same for accomplishing our second task, getting the posterior draws for the actor-level estimates from the cross-classified `b12.8` model, averaging over the levels of `block`.

```
s2 <-
b12.8 %>%
  spread_draws(b_Intercept, r_actor[actor,]) %>%
  mutate(p = (b_Intercept + r_actor) %>% inv_logit_scaled()) %>%
  select(.draw, actor, p) %>%
  ungroup() %>%
  mutate(actor = str_c("chimp ", actor, "'s average probability of pulling left")) %>%
  spread(value = p, key = actor)

str(s2)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 8 variables:
##   $ .draw : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.403 0.437 0.458 0.421 0.416 ...
##   $ chimp 2's average probability of pulling left: num 0.993 0.99 0.991 1 0.999 ...
##   $ chimp 3's average probability of pulling left: num 0.406 0.419 0.385 0.343 0.384 ...
##   $ chimp 4's average probability of pulling left: num 0.353 0.347 0.31 0.286 0.429 ...
##   $ chimp 5's average probability of pulling left: num 0.391 0.388 0.352 0.478 0.36 ...
##   $ chimp 6's average probability of pulling left: num 0.672 0.683 0.646 0.634 0.633 ...
##   $ chimp 7's average probability of pulling left: num 0.904 0.896 0.889 0.927 0.897 ...
```

To accomplish our third task, we augment the `spread_draws()` and first `mutate()` lines, and add a `filter()` line between them.

```
s3 <-
b12.8 %>%
  spread_draws(b_Intercept, r_actor[actor], r_block[block,]) %>%
  filter(block == 1) %>%
  mutate(p = (b_Intercept + r_actor + r_block) %>% inv_logit_scaled()) %>%
  select(.draw, actor, p) %>%
  ungroup() %>%
  mutate(actor = str_c("chimp ", actor, "'s average probability of pulling left")) %>%
  spread(value = p, key = actor)
```

Adding missing grouping variables: `block`

```
str(s3)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 16000 obs. of 9 variables:
##   $ block : int 1 1 1 1 1 1 1 1 1 ...
##   $ .draw : int 1 2 3 4 5 6 7 8 9 10 ...
##   $ chimp 1's average probability of pulling left: num 0.278 0.468 0.457 0.422 0.408 ...
##   $ chimp 2's average probability of pulling left: num 0.987 0.991 0.991 1 0.999 ...
##   $ chimp 3's average probability of pulling left: num 0.28 0.45 0.384 0.343 0.376 ...
##   $ chimp 4's average probability of pulling left: num 0.237 0.376 0.309 0.287 0.421 ...
##   $ chimp 5's average probability of pulling left: num 0.268 0.417 0.351 0.479 0.353 ...
##   $ chimp 6's average probability of pulling left: num 0.538 0.709 0.646 0.635 0.625 ...
##   $ chimp 7's average probability of pulling left: num 0.843 0.907 0.889 0.927 0.894 ...
```

Hopefully working through these examples gave you some insight on the relation between fixed and random effects within multilevel models, and perhaps added to your posterior-iteration-wrangling toolkit.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
```

```
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods   base
##
## other attached packages:
## [1] tidybayes_1.0.4    gridExtra_2.3       bayesplot_1.6.0     ggthemes_4.0.1
## [5] forcats_0.3.0     stringr_1.4.0       dplyr_0.8.0.1      purrr_0.2.5
## [9] readr_1.1.1       tidyr_0.8.1        tibble_2.1.1       tidyverse_1.2.1
## [13] brms_2.8.8       Rcpp_1.0.1         rstan_2.18.2      StanHeaders_2.18.0-1
## [17] ggplot2_3.1.1
##
## loaded via a namespace (and not attached):
## [1] nlme_3.1-137          matrixStats_0.54.0    xts_0.10-2
## [4] lubridate_1.7.4        threejs_0.3.1         httr_1.3.1
## [7] rprojroot_1.3-2       tools_3.5.1           backports_1.1.4
## [10] utf8_1.1.4            R6_2.3.0              DT_0.4
## [13] lazyeval_0.2.2        colorspace_1.3-2      withr_2.1.2
## [16] tidyselect_0.2.5      prettyunits_1.0.2     processx_3.2.1
## [19] Brobdingnag_1.2-6     compiler_3.5.1       rvest_0.3.2
## [22] cli_1.0.1            arrayhelpers_1.0-20160527 xml2_1.2.0
## [25] shinyjs_1.0           labeling_0.3          colourpicker_1.0
## [28] bookdown_0.9          scales_1.0.0          dygraphs_1.1.1.5
## [31] mvtnorm_1.0-10        ggridges_0.5.0       callr_3.1.0
## [34] digest_0.6.18         rmarkdown_1.10        base64enc_0.1-3
## [37] pkgconfig_2.0.2       htmltools_0.3.6      readxl_1.1.0
## [40] htmlwidgets_1.2        rlang_0.3.4          rstudioapi_0.7
## [43] shiny_1.1.0            svUnit_0.7-12        generics_0.0.2
## [46] zoo_1.8-2             jsonlite_1.5          crosstalk_1.0.0
## [49] gtools_3.8.1          inline_0.3.15        magrittr_1.5
## [52] loo_2.1.0              Matrix_1.2-14        fansi_0.4.0
## [55] munsell_0.5.0          abind_1.4-5          stringi_1.4.3
## [58] yaml_2.1.19            MASS_7.3-50          ggstance_0.3
## [61] pkgbuild_1.0.2         plyr_1.8.4           grid_3.5.1
## [64] promises_1.0.1         crayon_1.3.4         miniUI_0.1.1.1
## [67] lattice_0.20-35        haven_1.1.2          hms_0.4.2
## [70] knitr_1.20              ps_1.2.1             pillar_1.3.1
## [73] igraph_1.2.1           markdown_0.8          shinystan_2.5.0
## [76] reshape2_1.4.3          stats4_3.5.1         rstantools_1.5.1
## [79] glue_1.3.1.9000         evaluate_0.10.1      modelr_0.1.2
## [82] httpuv_1.4.4.2          cellranger_1.1.0     gtable_0.3.0
## [85] assertthat_0.2.0         xfun_0.3              mime_0.5
## [88] xtable_1.8-2            broom_0.5.1          coda_0.19-2
## [91] later_0.7.3             rsconnect_0.8.8      shinythemes_1.1.1
## [94] bridgesampling_0.6-0
```


Chapter 13

Adventures in Covariance

In this chapter, you'll see how to... specify varying slopes in combination with the varying intercepts of the previous chapter. This will enable pooling that will improve estimates of how different units respond to or are influenced by predictor variables. It will also improve estimates of intercepts, by borrowing information across parameter types. Essentially, varying slopes models are massive interaction machines. They allow every unit in the data to have its own unique response to any treatment or exposure or event, while also improving estimates via pooling. When the variation in slopes is large, the average slope is of less interest. Sometimes, the pattern of variation in slopes provides hints about omitted variables that explain why some units respond more or less. We'll see an example in this chapter.

The machinery that makes such complex varying effects possible will be used later in the chapter to extend the varying effects strategy to more subtle model types, including the use of continuous categories, using Gaussian process. (p. 388)

13.1 Varying slopes by construction

How should the robot pool information across intercepts and slopes? By modeling the joint population of intercepts and slopes, which means by modeling their covariance. In conventional multilevel models, the device that makes this possible is a joint multivariate Gaussian distribution for all of the varying effects, both intercepts and slopes. So instead of having two independent Gaussian distributions of intercepts and of slopes, the robot can do better by assigning a two-dimensional Gaussian distribution to both the intercepts (first dimension) and the slopes (second dimension). (p. 389)

In the **Rethinking: Why Gaussian?** box, McElreath discussed how researchers might use other multivariate distributions to model multiple random effects. The only one he named as an alternative to the Gaussian was the multivariate Student's t . As it turns out, brms does currently allow users to use multivariate Student's t in this way. For details, check out [this discussion in the brms GitHub page](#). Bürkner's exemplar syntax from his comment on May 13, 2018, was `y ~ x + (x | gr(g, dist = "student"))`. I haven't experimented with this, but if you do, do consider [commenting on how it went](#).

13.1.1 Simulate the population.

If you follow this section closely, it's a great template for simulating multilevel code for any of your future projects. You might think of this as an alternative to a frequentist power analysis. Vourre has done [some nice work along these lines](#), too.

```
a      <- 3.5 # average morning wait time
b      <- -1 # average difference afternoon wait time
sigma_a <- 1 # std dev in intercepts
sigma_b <- 0.5 # std dev in slopes
rho    <- -.7 # correlation between intercepts and slopes

# the next three lines of code simply combine the terms, above
mu     <- c(a, b)
cov_ab <- sigma_a * sigma_b * rho
```

```
sigma <- matrix(c(sigma_a^2, cov_ab,
                    cov_ab, sigma_b^2), ncol = 2)
```

If you haven't used `matrix()` before, you might get a sense of the elements like so.

```
matrix(c(1, 2,
        3, 4), nrow = 2, ncol = 2)

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

This next block of code will finally yield our café data.

```
library(tidyverse)

sigmas <- c(sigma_a, sigma_b)          # standard deviations
rho     <- matrix(c(1, rho,             # correlation matrix
                    rho, 1), nrow = 2)

# now matrix multiply to get covariance matrix
sigma <- diag(sigmas) %*% rho %*% diag(sigmas)

# how many cafes would you like?
n_cafes <- 20

set.seed(13) # used to replicate example
vary_effects <-
  MASS::mvrnorm(n_cafes, mu, sigma) %>%
  data.frame() %>%
  set_names("a_cafe", "b_cafe")

head(vary_effects)

##      a_cafe      b_cafe
## 1  2.917639 -0.8649154
## 2  3.552770 -1.6814372
## 3  1.694390 -0.4168858
## 4  3.442417 -0.6011724
## 5  2.289988 -0.7461953
## 6  3.069283 -0.8839639
```

Let's make sure we're keeping this all straight. `a_cafe` = our café-specific intercepts; `b_cafe` = our café-specific slopes. These aren't the actual data, yet. But at this stage, it might make sense to ask *What's the distribution of `a_cafe` and `b_cafe`?* Our variant of Figure 13.2 contains the answer.

For our plots in this chapter, we'll use a custom theme. The color palette will come from the "pearl_earring" palette of the `dutchmasters` package. You can learn more about the original painting, Vermeer's *Girl with a Pearl Earring*, [here](#).

```
# devtools::install_github("EdwinTh/dutchmasters")
library(dutchmasters)

dutchmasters$pearl_earring

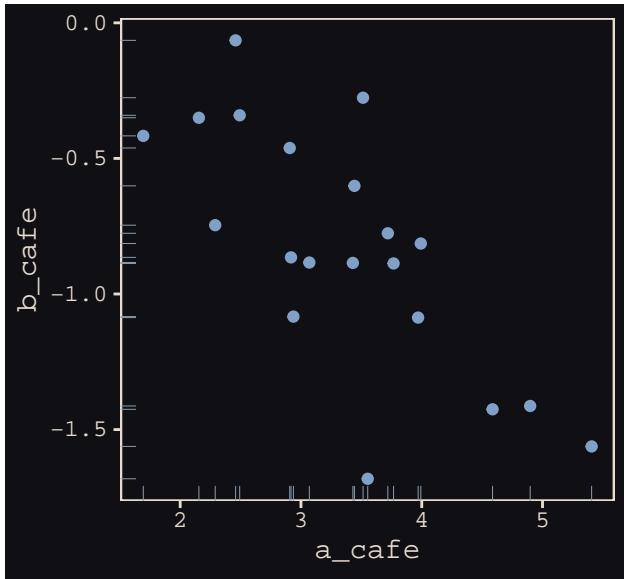
##      red(lips)      skin      blue(scarf1)      blue(scarf2)      white(collar)
##      "#A65141"      "#E7CDC2"      "#80A0C7"      "#394165"      "#FCF9F0"
##      gold(dress)    gold(dress2) black(background) grey(scarf3)    yellow(scarf4)
##      "#B1934A"      "#DCA258"      "#100F14"      "#8B9DAF"      "#EEDA9D"
##
##      "#E8DCCF"
```

We'll name our custom theme `theme_pearl_earring`.

```
theme_pearl_earring <-
  theme(text      = element_text(color = "#E8DCCF", family = "Courier"),
        strip.text = element_text(color = "#E8DCCF", family = "Courier"),
        axis.text  = element_text(color = "#E8DCCF"),
        axis.ticks = element_line(color = "#E8DCCF"),
        line       = element_line(color = "#E8DCCF"),
        plot.background = element_rect(fill = "#100F14", color = "transparent"),
        panel.background = element_rect(fill = "#100F14", color = "#E8DCCF"),
        strip.background = element_rect(fill = "#100F14", color = "transparent"),
        panel.grid = element_blank(),
        legend.background = element_rect(fill = "#100F14", color = "transparent"),
        legend.key     = element_rect(fill = "#100F14", color = "transparent"),
        axis.line = element_blank())
```

Now we're ready to plot Figure 13.2.

```
vary_effects %>%
  ggplot(aes(x = a_cafe, y = b_cafe)) +
  geom_point(color = "#80A0C7") +
  geom_rug(color = "#8B9DAF", size = 1/7) +
  theme_pearl_earring
```



Again, these are not “data.” Figure 13.2 shows a distribution of *parameters*.

13.1.2 Simulate observations.

Here we put those simulated parameters to use.

```
n_visits <- 10
sigma     <- 0.5 # std dev within cafes

set.seed(13) # used to replicate example
d <-
  vary_effects %>%
  mutate(cafe      = 1:n_cafes) %>%
  expand(nesting(cafe, a_cafe, b_cafe), visit = 1:n_visits) %>%
  mutate(afternoon = rep(0:1, times = n() / 2)) %>%
  mutate(mu        = a_cafe + b_cafe * afternoon) %>%
  mutate(wait      = rnorm(n = n(), mean = mu, sd = sigma))
```

We might peek at the data.

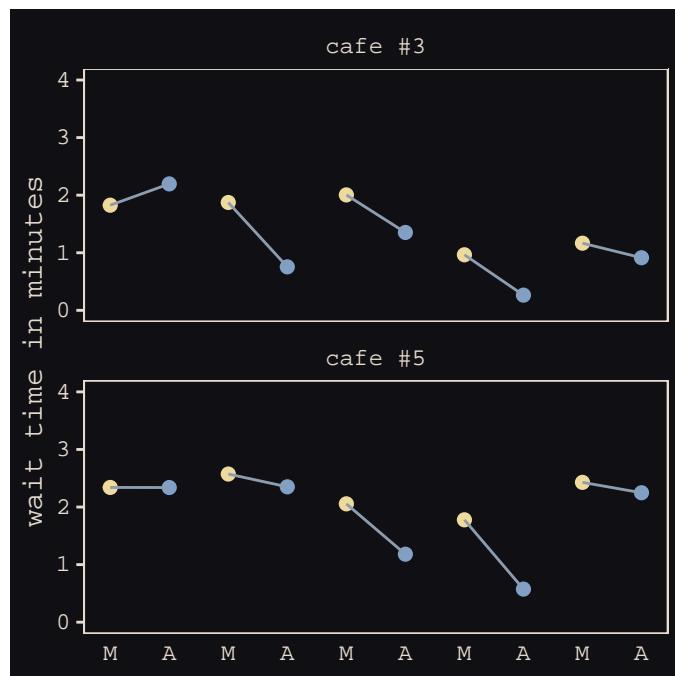
```
d %>%
  head()

## # A tibble: 6 x 7
##   cafe a_cafe b_cafe visit afternoon mu   wait
##   <int> <dbl> <dbl> <int>     <int> <dbl> <dbl>
## 1     1    2.92 -0.865     1         0    2.92  3.19
## 2     1    2.92 -0.865     2         1    2.05  1.91
## 3     1    2.92 -0.865     3         0    2.92  3.81
## 4     1    2.92 -0.865     4         1    2.05  2.15
## 5     1    2.92 -0.865     5         0    2.92  3.49
## 6     1    2.92 -0.865     6         1    2.05  2.26
```

Now we've finally simulated our data, we are ready to make our version of Figure 13.1, from way back on page 388.

```
d %>%
  mutate(afternoon = ifelse(afternoon == 0, "M", "A"),
        day      = rep(rep(1:5, each = 2), times = n_cafes)) %>%
  filter(cafe %in% c(3, 5)) %>%
  mutate(cafe = ifelse(cafe == 3, "cafe #3", "cafe #5")) %>%

  ggplot(aes(x = visit, y = wait, group = day)) +
  geom_point(aes(color = afternoon), size = 2) +
  geom_line(color = "#8B9DAF") +
  scale_color_manual(values = c("#80A0C7", "#EEDA9D")) +
  scale_x_continuous(NULL, breaks = 1:10,
                     labels = rep(c("M", "A"), times = 5)) +
  coord_cartesian(ylim = 0:4) +
  ylab("wait time in minutes") +
  theme_pearl_earring +
  theme(legend.position = "none",
        axis.ticks.x = element_blank()) +
  facet_wrap(~cafe, ncol = 1)
```



13.1.3 The varying slopes model.

The statistical formula for our varying-slopes model follows the form

$$\begin{aligned}
 \text{wait}_i &\sim \text{Normal}(\mu_i, \sigma) \\
 \mu_i &= \alpha_{\text{cafe}_i} + \beta_{\text{cafe}_i} \text{afternoon}_i \\
 \begin{bmatrix} \alpha_{\text{cafe}} \\ \beta_{\text{cafe}} \end{bmatrix} &\sim \text{MVNormal}\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \mathbf{S}\right) \\
 \mathbf{S} &= \begin{pmatrix} \sigma_\alpha & 0 \\ 0 & \sigma_\beta \end{pmatrix} \mathbf{R} \begin{pmatrix} \sigma_\alpha & 0 \\ 0 & \sigma_\beta \end{pmatrix} \\
 \alpha &\sim \text{Normal}(0, 10) \\
 \beta &\sim \text{Normal}(0, 10) \\
 \sigma &\sim \text{HalfCauchy}(0, 1) \\
 \sigma_\alpha &\sim \text{HalfCauchy}(0, 1) \\
 \sigma_\beta &\sim \text{HalfCauchy}(0, 1) \\
 \mathbf{R} &\sim \text{LKJcorr}(2)
 \end{aligned}$$

Of the notable new parts, \mathbf{S} is the covariance matrix and \mathbf{R} is the corresponding correlation matrix, which we might more fully express as

$$\begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

And according to our prior, \mathbf{R} is distributed as $\text{LKJcorr}(2)$. We'll use `rethinking::rlkjcorr()` to get a better sense of what that even is.

```

library(rethinking)

n_sim <- 1e5

set.seed(13)
r_1 <-
  rlkjcorr(n_sim, K = 2, eta = 1) %>%
  as_tibble()

set.seed(13)
r_2 <-
  rlkjcorr(n_sim, K = 2, eta = 2) %>%
  as_tibble()

set.seed(13)
r_4 <-
  rlkjcorr(n_sim, K = 2, eta = 4) %>%
  as_tibble()

```

Here are the LKJcorr distributions of Figure 13.3.

```

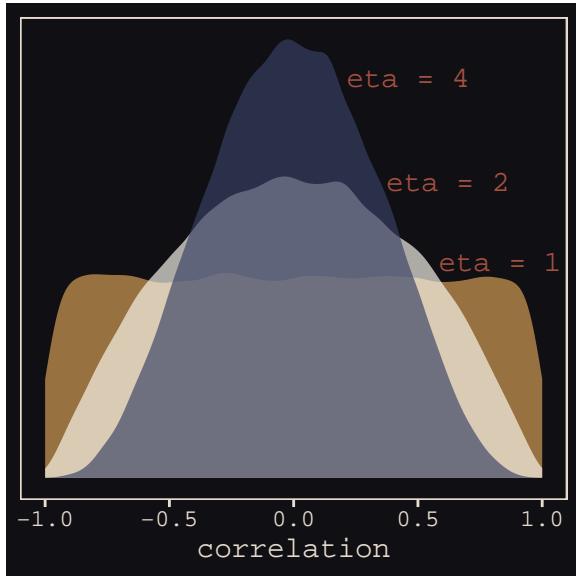
ggplot(data = r_1, aes(x = V2)) +
  geom_density(color = "transparent", fill = "#DCA258", alpha = 2/3) +
  geom_density(data = r_2,
              color = "transparent", fill = "#FCF9F0", alpha = 2/3) +
  geom_density(data = r_4,
              color = "transparent", fill = "#394165", alpha = 2/3) +
  geom_text(data = tibble(x      = c(.83, .62, .46),
                         y      = c(.54, .74, 1),

```

```

      label = c("eta = 1", "eta = 2", "eta = 4")),
      aes(x = x, y = y, label = label),
      color = "#A65141", family = "Courier") +
scale_y_continuous(NULL, breaks = NULL) +
xlab("correlation") +
theme_pearl_earring

```



Okay, let's get ready to model and switch out rethinking for brms.

```

detach(package:rethinking, unload = T)
library(brms)

```

As defined above, our first model has both varying intercepts and `afternoon` slopes. I should point out that the `(1 + afternoon | cafe)` syntax specifies that we'd like `brm()` to fit the random effects for `1` (i.e., the intercept) and the `afternoon` slope as correlated. Had we wanted to fit a model in which they were orthogonal, we'd have coded `(1 + afternoon || cafe)`.

```

b13.1 <-
  brm(data = d, family = gaussian,
    wait ~ 1 + afternoon + (1 + afternoon | cafe),
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 2), class = sd),
              prior(cauchy(0, 2), class = sigma),
              prior(lkj(2), class = cor)),
    iter = 5000, warmup = 2000, chains = 2, cores = 2,
    seed = 13)

```

With Figure 13.4, we assess how the posterior for the correlation of the random effects compares to its prior.

```

post <- posterior_samples(b13.1)

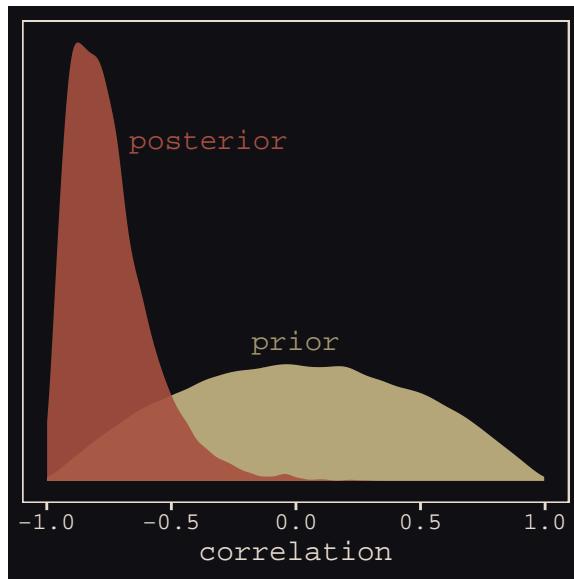
post %>%
  ggplot(aes(x = cor_cafe__Intercept__afternoon)) +
  geom_density(data = r_2, aes(x = V2),
               color = "transparent", fill = "#EEDA9D", alpha = 3/4) +
  geom_density(color = "transparent", fill = "#A65141", alpha = 9/10) +
  annotate("text", label = "posterior",
          x = -0.35, y = 2.2,
          color = "#A65141", family = "Courier") +
  annotate("text", label = "prior",
          x = 0.35, y = 2.2,
          color = "#A65141", family = "Courier")

```

```

x = 0, y = 0.9,
color = "#EEDA9D", alpha = 2/3, family = "Courier") +
scale_y_continuous(NULL, breaks = NULL) +
xlab("correlation") +
theme_pearl_earring

```



McElreath then depicted multidimensional shrinkage by plotting the posterior mean of the varying effects compared to their raw, unpooled estimated. With brms, we can get the `cafe`-specific intercepts and `afternoon` slopes with `coef()`, which returns a three-dimensional list.

```

# coef(b13.1) %>% glimpse()
coef(b13.1)

```

```

## $cafe
## , , Intercept
##
##   Estimate Est.Error    Q2.5    Q97.5
## 1  3.257478 0.2035942 2.852822 3.656762
## 2  3.165662 0.2087660 2.763492 3.579353
## 3  1.615157 0.2060121 1.222783 2.016879
## 4  3.335387 0.1964793 2.949353 3.717637
## 5  2.296397 0.2063222 1.887291 2.698056
## 6  3.148887 0.2017595 2.740449 3.536169
## 7  2.648014 0.2090663 2.241426 3.053719
## 8  3.642193 0.1988416 3.245539 4.027335
## 9  4.117556 0.2028759 3.724034 4.515102
## 10 2.322228 0.2080664 1.910779 2.728812
## 11 4.444412 0.2075830 4.037092 4.841307
## 12 2.833985 0.2002205 2.439787 3.228190
## 13 4.653762 0.2050913 4.249860 5.054741
## 14 5.535245 0.2214844 5.115992 5.971740
## 15 3.561178 0.1994760 3.170220 3.949092
## 16 3.485721 0.1980807 3.093399 3.871864
## 17 2.341366 0.1987429 1.943998 2.730567
## 18 3.959822 0.2078971 3.560317 4.367699
## 19 3.500621 0.1999719 3.104086 3.895698
## 20 3.168557 0.2026318 2.764338 3.566792
##
## , , afternoon

```

```

##          Estimate   Est.Error      Q2.5     Q97.5
## 1 -0.9075166 0.2096445 -1.3618266 -0.511335490
## 2 -1.0466650 0.2429569 -1.5658734 -0.631545880
## 3 -0.4292542 0.2328915 -0.8855643  0.024001165
## 4 -0.7421363 0.2071748 -1.1324862 -0.311491345
## 5 -0.5481318 0.2163228 -0.9617509 -0.120817780
## 6 -0.7952144 0.2086028 -1.2129496 -0.368283468
## 7 -0.4905239 0.2285836 -0.8967930 -0.007445871
## 8 -0.8224418 0.2090859 -1.2171772 -0.390551849
## 9 -1.0802684 0.2129073 -1.5259491 -0.678236588
## 10 -0.4258400 0.2287290 -0.8410538  0.062666711
## 11 -1.0932314 0.2210538 -1.5331957 -0.660440952
## 12 -0.7616165 0.2098765 -1.1865653 -0.360890306
## 13 -1.2126147 0.2234728 -1.6645061 -0.787262410
## 14 -1.5344024 0.2712118 -2.0737942 -1.023297217
## 15 -0.9242634 0.2080521 -1.3465731 -0.521195971
## 16 -0.7348618 0.2107951 -1.1228306 -0.296388693
## 17 -0.5667775 0.2138634 -0.9935640 -0.132124851
## 18 -1.0093023 0.2111856 -1.4437750 -0.605782458
## 19 -0.7444063 0.2116041 -1.1260144 -0.291720917
## 20 -0.7314524 0.2081339 -1.1390389 -0.296033163

```

Here's the code to extract the relevant elements from the `coef()` list, convert them to a tibble, and add the `cafe` index.

```

partially_pooled_params <-
  # with this line we select each of the 20 cafe's posterior mean (i.e., Estimate)
  # for both `Intercept` and `afternoon`
  coef(b13.1)$cafe[, 1, 1:2] %>%
  as_tibble() %>%
    # convert the two vectors to a tibble
  rename(Slope = afternoon) %>%
  mutate(cafe = 1:nrow(.)) %>% # add the `cafe` index
  select(cafe, everything())    # simply moving `cafe` to the leftmost position

```

Like McElreath, we'll compute the unpooled estimates directly from the data.

```

# compute unpooled estimates directly from data
un_pooled_params <-
  d %>%
  # with these two lines, we compute the mean value for each cafe's wait time
  # in the morning and then the afternoon
  group_by(afternoon, cafe) %>%
  summarise(mean = mean(wait)) %>%
  ungroup() %>% # ungrouping allows us to alter afternoon, one of the grouping variables
  mutate(afternoon = ifelse(afternoon == 0, "Intercept", "Slope")) %>%
  spread(key = afternoon, value = mean) %>% # use `spread()` just as in the previous block
  mutate(Slope = Slope - Intercept)           # finally, here's our slope!

# here we combine the partially-pooled and unpooled means into a single data object,
# which will make plotting easier.
params <-
  # `bind_rows()` will stack the second tibble below the first
  bind_rows(partially_pooled_params, un_pooled_params) %>%
  # index whether the estimates are pooled
  mutate(pooled = rep(c("partially", "not"), each = nrow(.)/2))

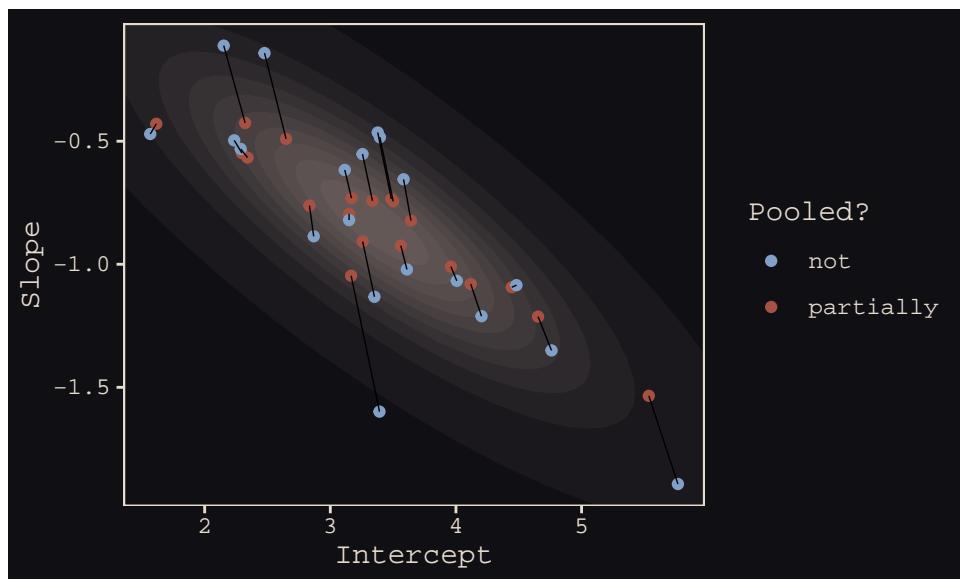
# here's a glimpse at what we've been working for
params %>%
  slice(c(1:5, 36:40))

```

```
## # A tibble: 10 x 4
##   cafe Intercept Slope pooled
##   <int>     <dbl>  <dbl> <chr>
## 1     1      3.26 -0.908 partially
## 2     2      3.17 -1.05  partially
## 3     3      1.62 -0.429 partially
## 4     4      3.34 -0.742 partially
## 5     5      2.30 -0.548 partially
## 6    16      3.38 -0.465 not
## 7    17      2.29 -0.531 not
## 8    18      4.01 -1.07  not
## 9    19      3.39 -0.484 not
## 10   20      3.12 -0.617 not
```

Finally, here's our code for Figure 13.5.a, showing shrinkage in two dimensions.

```
ggplot(data = params, aes(x = Intercept, y = Slope)) +
  stat_ellipse(geom = "polygon", type = "norm", level = 1/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 2/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 3/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 4/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 5/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 6/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 7/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 8/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = 9/10, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  stat_ellipse(geom = "polygon", type = "norm", level = .99, size = 0, alpha = 1/20, fill = "#E7CDC2") +
  geom_point(aes(group = cafe, color = pooled)) +
  geom_line(aes(group = cafe), size = 1/4) +
  scale_color_manual("Pooled?", values = c("#80A0C7", "#A65141")) +
  coord_cartesian(xlim = range(params$Intercept),
                  ylim = range(params$Slope)) +
  theme_pearl_earring
```



Learn more about `stat_ellipse()`, [here](#). Let's prep for Figure 13.5.b.

```
# retrieve the partially-pooled estimates with `coef()`
partially_pooled_estimates <-
  coef(b13.1)$cafe[ , 1, 1:2] %>%
  # convert the two vectors to a tibble
```

```

as_tibble() %>%
# the Intercept is the wait time for morning (i.e., `afternoon == 0`)
rename(morning = Intercept) %>%
# `afternoon` wait time is the `morning` wait time plus the afternoon slope
mutate(afternoon = morning + afternoon,
      cafe      = 1:n()) %>% # add the `cafe` index
select(cafe, everything())

# compute unpooled estimates directly from data
un_pooled_estimates <-
d %>%
# as above, with these two lines, we compute each cafe's mean wait value by time of day
group_by(afternoon, cafe) %>%
summarise(mean = mean(wait)) %>%
# ungrouping allows us to alter the grouping variable, afternoon
ungroup() %>%
mutate(afternoon = ifelse(afternoon == 0, "morning", "afternoon")) %>%
# this separates out the values into morning and afternoon columns
spread(key = afternoon, value = mean)

estimates <-
bind_rows(partially_pooled_estimates, un_pooled_estimates) %>%
mutate(pooled = rep(c("partially", "not"), each = n() / 2))

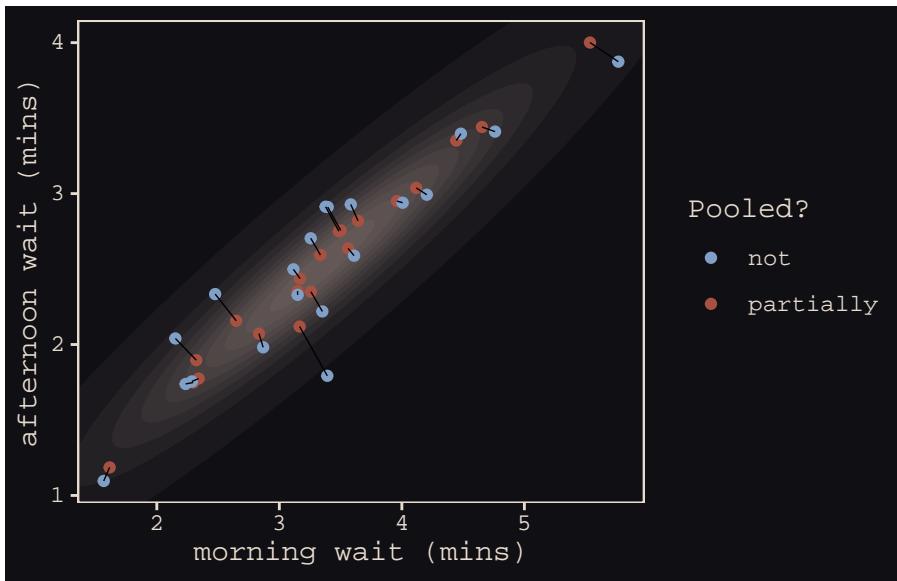
```

The code for Figure 13.5.b.

```

ggplot(data = estimates, aes(x = morning, y = afternoon)) +
# nesting `stat_ellipse()` within `mapply()` is a less redundant way to produce the
# ten-layered semitransparent ellipses we did with ten lines of `stat_ellipse()`
# functions in the previous plot
mapply(function(level) {
  stat_ellipse(geom = "polygon", type = "norm",
               size = 0, alpha = 1/20, fill = "#E7CDC2",
               level = level)
},
# Enter the levels here
level = c(seq(from = 1/10, to = 9/10, by = 1/10), .99)) +
geom_point(aes(group = cafe, color = pooled)) +
geom_line(aes(group = cafe), size = 1/4) +
scale_color_manual("Pooled?",
                  values = c("#80A0C7", "#A65141")) +
coord_cartesian(xlim = range(estimates$morning),
                ylim = range(estimates$afternoon)) +
labs(x = "morning wait (mins)",
     y = "afternoon wait (mins)") +
theme_pearl_earring

```



13.2 Example: Admission decisions and gender

Let's revisit the infamous UCB admissions data.

```
library(rethinking)
data(UCBadmit)
d <- UCBadmit
```

Here we detach rethinking, reload brms, and augment the data a bit.

```
detach(package:rethinking, unload = T)
library(brms)
rm(UCBadmit)

d <-
d %>%
  mutate(male      = ifelse(applicant.gender == "male", 1, 0),
     dept_id = rep(1:6, each = 2))
```

13.2.1 Varying intercepts.

The statistical formula for our varying-intercepts logistic regression model follows the form

$$\begin{aligned} \text{admit}_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{dept_id}_i} + \beta \text{male}_i \\ \alpha_{\text{dept_id}} &\sim \text{Normal}(\alpha, \sigma) \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{HalfCauchy}(0, 2) \end{aligned}$$

Since there's only one left-hand term in our `(1 | dept_id)` code, there's only one random effect.

```
b13.2 <-
  brm(data = d, family = binomial,
    admit | trials(applications) ~ 1 + male + (1 | dept_id),
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(cauchy(0, 2), class = sd)),
    iter = 4500, warmup = 500, chains = 3, cores = 3,
    seed = 13,
    control = list(adapt_delta = 0.99))
```

Since we don't have a `depth=2` argument in `brms::summary()`, we'll have to get creative. One way to look at the parameters is with `b13.2$fit`:

```
b13.2$fit
```

```
## Inference for Stan model: bc63eecdab9ca7a1bd98aff8a3c74d8d2.
## 3 chains, each with iter=4500; warmup=500; thin=1;
## post-warmup draws per chain=4000, total post-warmup draws=12000.
##
##                mean se_mean   sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## b_Intercept     -0.58    0.01 0.65 -1.87 -0.94 -0.58 -0.21  0.76 1894    1
## b_male          -0.10    0.00 0.08 -0.26 -0.15 -0.10 -0.04  0.07 4728    1
## sd_dept_id__Intercept 1.48    0.01 0.58  0.77  1.10  1.35  1.70  2.91 1814    1
## r_dept_id[1,Intercept] 1.25    0.01 0.65 -0.07  0.88  1.26  1.62  2.55 1913    1
## r_dept_id[2,Intercept] 1.21    0.01 0.65 -0.10  0.83  1.21  1.58  2.53 1914    1
## r_dept_id[3,Intercept] 0.00    0.01 0.65 -1.34 -0.37 -0.01  0.36  1.27 1899    1
## r_dept_id[4,Intercept] -0.04   0.01 0.65 -1.36 -0.40 -0.04  0.33  1.25 1903    1
## r_dept_id[5,Intercept] -0.48   0.01 0.65 -1.83 -0.85 -0.48 -0.11  0.81 1909    1
## r_dept_id[6,Intercept] -2.03   0.01 0.66 -3.38 -2.40 -2.03 -1.64 -0.72 1959    1
## lp__            -61.84   0.05 2.51 -67.60 -63.34 -61.50 -60.02 -57.92 2635    1
##
## Samples were drawn using NUTS(diag_e) at Sat Apr 20 14:18:19 2019.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

However, notice that the group-specific parameters don't match up with those in the text. Though our `r_dept_id[1,Intercept]` had a posterior mean of 1.25, the number for `a_dept[1]` in the text is 0.67. This is because the `brms` package presented the random effects in the **non-centered** metric. The `rethinking` package, in contrast, presented the random effects in the **centered** metric. On page 399, McElreath wrote:

Remember, the values above are the α_{DEPT} estimates, and so they are deviations from the global mean α , which in this case has posterior mean -0.58. So department A, "[1]" in the table, has the highest average admission rate. Department F, "[6]" in the table, has the lowest.

Here's another fun fact:

```
# numbers taken from the mean column on page 399 in the text
c(0.67, 0.63, -0.59, -0.62, -1.06, -2.61) %>% mean()
```

```
## [1] -0.5966667
```

The average of the `rethinking`-based **centered** random effects is within rounding error of the global mean, -0.58. If you want the random effects in the **centered** metric from `brms`, you can use the `coef()` function:

```
coef(b13.2)
```

```
## $dept_id
## , , Intercept
##
##      Estimate   Est.Error     Q2.5     Q97.5
## 1  0.6761368 0.09775346  0.4875966  0.8692484
## 2  0.6313927 0.11483278  0.4126650  0.8608403
## 3 -0.5820628 0.07404136 -0.7291298 -0.4391695
## 4 -0.6149658 0.08624727 -0.7851875 -0.4490055
## 5 -1.0597847 0.09848451 -1.2579206 -0.8649718
## 6 -2.6064264 0.15776514 -2.9211817 -2.3091631
##
## , , male
##
##      Estimate   Est.Error     Q2.5     Q97.5
## 1 -0.09643949 0.08121842 -0.2619235  0.06542184
## 2 -0.09643949 0.08121842 -0.2619235  0.06542184
## 3 -0.09643949 0.08121842 -0.2619235  0.06542184
## 4 -0.09643949 0.08121842 -0.2619235  0.06542184
## 5 -0.09643949 0.08121842 -0.2619235  0.06542184
## 6 -0.09643949 0.08121842 -0.2619235  0.06542184
```

And just to confirm, the average of the posterior means of the `Intercept` random effects with `brms::coef()` is also the global mean within rounding error:

```
mean(coef(b13.2)$dept_id[, "Estimate", "Intercept"])
```

```
## [1] -0.5926184
```

Note how `coef()` returned a three-dimensional list.

```
coef(b13.2) %>% str()
```

```
## List of 1
## $ dept_id: num [1:6, 1:4, 1:2] 0.676 0.631 -0.582 -0.615 -1.06 ...
##   ..- attr(*, "dimnames")=List of 3
##     ...$ : chr [1:6] "1" "2" "3" "4" ...
##     ...$ : chr [1:4] "Estimate" "Est.Error" "Q2.5" "Q97.5"
##     ...$ : chr [1:2] "Intercept" "male"
```

If you just want the parameter summaries for the random intercepts, you have to use three-dimensional indexing.

```
coef(b13.2)$dept_id[, , "Intercept"] # this also works: coef(b13.2)$dept_id[, , 1]
```

```
##      Estimate   Est.Error     Q2.5     Q97.5
## 1  0.6761368 0.09775346  0.4875966  0.8692484
## 2  0.6313927 0.11483278  0.4126650  0.8608403
## 3 -0.5820628 0.07404136 -0.7291298 -0.4391695
## 4 -0.6149658 0.08624727 -0.7851875 -0.4490055
## 5 -1.0597847 0.09848451 -1.2579206 -0.8649718
## 6 -2.6064264 0.15776514 -2.9211817 -2.3091631
```

So to get our `brms` summaries in a similar format to those in the text, we'll have to combine `coef()` with `fixef()` and `VarCorr()`.

```

rbind(coef(b13.2)$dept_id[, , "Intercept"],
      fixef(b13.2),
      VarCorr(b13.2)$dept_id$sd)

##           Estimate   Est.Error    Q2.5    Q97.5
## 1       0.67613683 0.09775346 0.4875966 0.86924836
## 2       0.63139265 0.11483278 0.4126650 0.86084032
## 3      -0.58206284 0.07404136 -0.7291298 -0.43916951
## 4      -0.61496585 0.08624727 -0.7851875 -0.44900551
## 5      -1.05978468 0.09848451 -1.2579206 -0.86497180
## 6      -2.60642640 0.15776514 -2.9211817 -2.30916314
## Intercept -0.57878250 0.64522749 -1.8721229 0.75534246
## male     -0.09643949 0.08121842 -0.2619235 0.06542184
## Intercept  1.47518830 0.57524043  0.7737629  2.90850757

```

And a little more data wrangling will make the summaries easier to read:

```

rbind(coef(b13.2)$dept_id[, , "Intercept"],
      fixef(b13.2),
      VarCorr(b13.2)$dept_id$sd) %>%
  as_tibble() %>%
  mutate(parameter = c(paste("Intercept [", 1:6, "] ", sep = ""),
                        "Intercept", "male", "sigma")) %>%
  select(parameter, everything()) %>%
  mutate_if(is_double, round, digits = 2)

```

```

## # A tibble: 9 x 5
##   parameter   Estimate Est.Error  Q2.5  Q97.5
##   <chr>        <dbl>     <dbl> <dbl> <dbl>
## 1 Intercept  0.68     0.1     0.49  0.87
## 2 Intercept  0.63     0.11    0.41  0.86
## 3 Intercept -0.580    0.07   -0.73 -0.44
## 4 Intercept -0.61     0.09   -0.79 -0.45
## 5 Intercept -1.06     0.1    -1.26 -0.86
## 6 Intercept -2.61     0.16   -2.92 -2.31
## 7 Intercept -0.580    0.65   -1.87  0.76
## 8 male       -0.1     0.08   -0.26  0.07
## 9 sigma      1.48     0.580  0.77  2.91

```

I'm not aware of a slick and easy way to get the `n_eff` and `Rhat` summaries into the mix. But if you're fine with working with the brms-default **non-centered** parameterization, `b13.2$fit` gets you those just fine.

One last thing. The `broom package` offers a very handy way to get those brms random effects. Just throw the model `brm()` fit into the `tidy()` function.

```

library(broom)

tidy(b13.2) %>%
  mutate_if(is.numeric, round, digits = 2) # this line just rounds the output

```

	term	estimate	std.error	lower	upper
## 1	b_Intercept	-0.58	0.65	-1.58	0.47
## 2	b_male	-0.10	0.08	-0.23	0.04
## 3	sd_dept_id_Intercept	1.48	0.58	0.84	2.49
## 4	r_dept_id[1,Intercept]	1.25	0.65	0.20	2.26
## 5	r_dept_id[2,Intercept]	1.21	0.65	0.17	2.22
## 6	r_dept_id[3,Intercept]	0.00	0.65	-1.06	1.01
## 7	r_dept_id[4,Intercept]	-0.04	0.65	-1.08	0.97

```
## 8 r_dept_id[5,Intercept] -0.48 0.65 -1.53 0.53
## 9 r_dept_id[6,Intercept] -2.03 0.66 -3.09 -1.00
## 10 lp__ -61.84 2.51 -66.41 -58.34
```

But note how, just as with `b13.2$fit`, this approach summarizes the posterior with the **non-centered** parameterization. Which is a fine parameterization. It's just a little different from what you'll get when using `precis(m13.2, depth=2)`, as in the text.

13.2.2 Varying effects of being male.

Now we're ready to allow our `male` dummy to varies, too, the statistical model follows the form

$$\begin{aligned} \text{admit}_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{dept_id}_i} + \beta_{\text{dept_id}_i} \text{male}_i \\ \begin{bmatrix} \alpha_{\text{dept_id}} \\ \beta_{\text{dept_id}} \end{bmatrix} &\sim \text{MVNormal}\left(\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \mathbf{S}\right) \\ \mathbf{S} &= \begin{pmatrix} \sigma_\alpha & 0 \\ 0 & \sigma_\beta \end{pmatrix} \mathbf{R} \begin{pmatrix} \sigma_\alpha & 0 \\ 0 & \sigma_\beta \end{pmatrix} \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \\ (\sigma_\alpha, \sigma_\beta) &\sim \text{HalfCauchy}(0, 2) \\ \mathbf{R} &\sim \text{LKJcorr}(2) \end{aligned}$$

Fit the model.

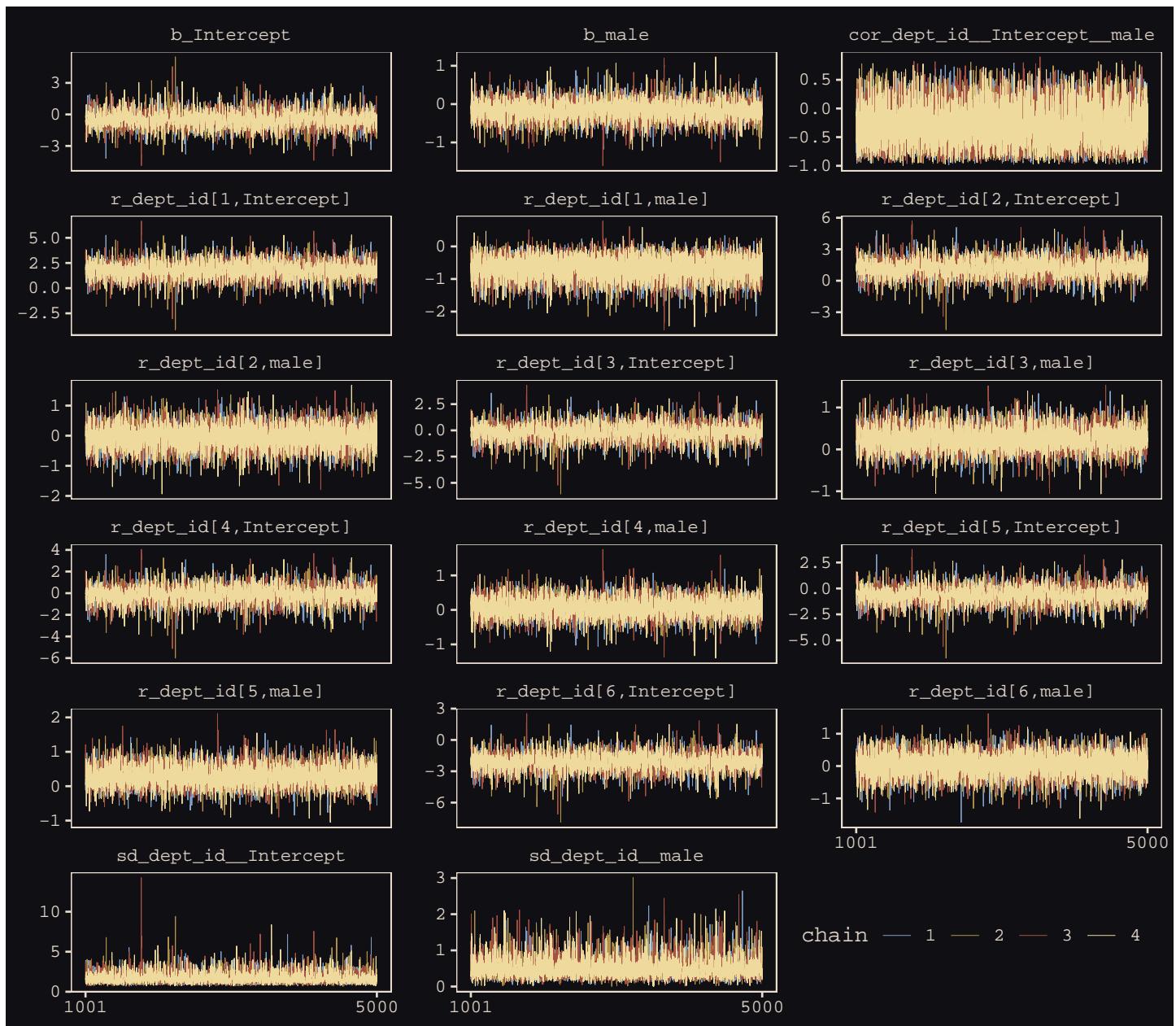
```
b13.3 <-
  brm(data = d, family = binomial,
    admit | trials(applications) ~ 1 + male + (1 + male | dept_id),
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(cauchy(0, 2), class = sd),
              prior(lkj(2), class = cor)),
    iter = 5000, warmup = 1000, chains = 4, cores = 4,
    seed = 13,
    control = list(adapt_delta = .99,
                  max_treedepth = 12))
```

McElreath encouraged us to make sure the chains look good. Instead of relying on convenience functions, let's do it by hand.

```
post <- posterior_samples(b13.3, add_chain = T)

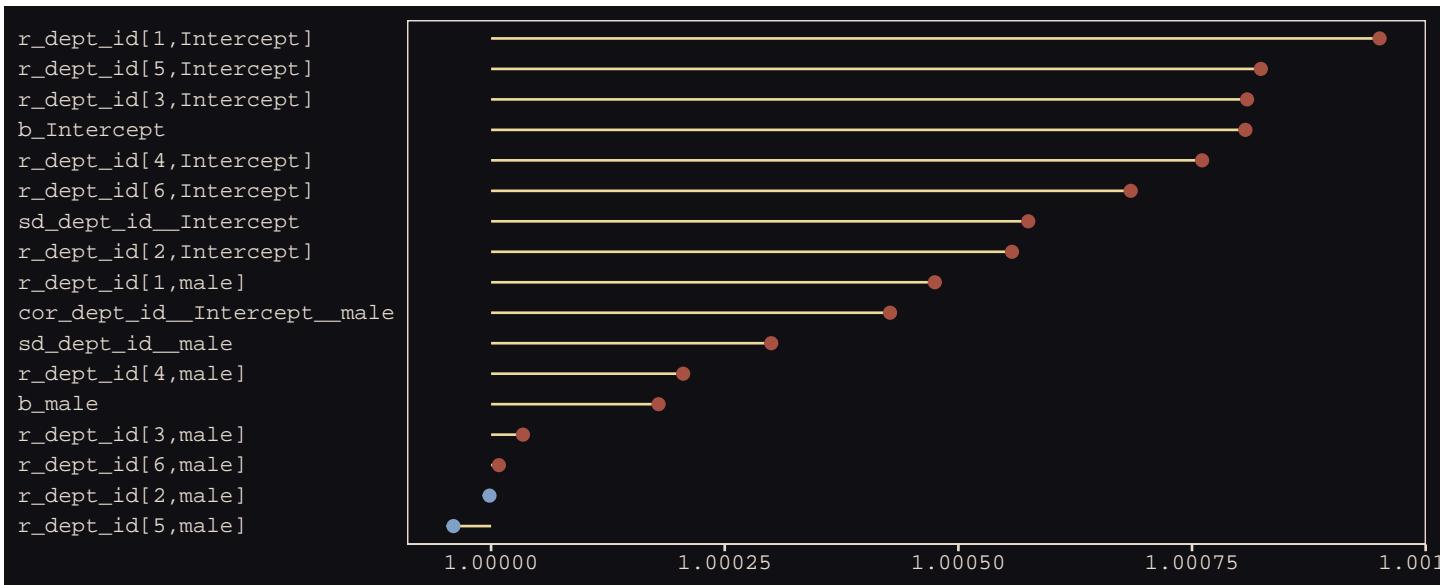
post %>%
  select(-lp__) %>%
  gather(key, value, -chain, -iter) %>%
  mutate(chain = as.character(chain)) %>%

  ggplot(aes(x = iter, y = value, group = chain, color = chain)) +
  geom_line(size = 1/15) +
  scale_color_manual(values = c("#80A0C7", "#B1934A", "#A65141", "#EEDA9D")) +
  scale_x_continuous(NULL, breaks = c(1001, 5000)) +
  ylab(NULL) +
  theme_pearl_earring +
  theme(legend.position = c(.825, .06),
        legend.direction = "horizontal") +
  facet_wrap(~key, ncol = 3, scales = "free_y")
```



Our chains look great. While we're at it, let's examine the \hat{R} values in a handmade plot, too.

```
rhat(b13.3) %>%
  data.frame() %>%
  rownames_to_column() %>%
  set_names("parameter", "rhat") %>%
  filter(parameter != "lp__") %>%
  ggplot(aes(x = rhat, y = reorder(parameter, rhat))) +
  geom_segment(aes(xend = 1, yend = parameter),
               color = "#EEDA9D") +
  geom_point(aes(color = rhat > 1),
             size = 2) +
  scale_color_manual(values = c("#80A0C7", "#A65141")) +
  labs(x = NULL, y = NULL) +
  theme_pearl_earring +
  theme(legend.position = "none",
        axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0))
```



There are some respectable \hat{R} values. The plot accentuates their differences, but they're all basically 1 (e.g., see what happens if you set `coord_cartesian(xlim = c(0.99, 1.01))`). Here are the random effects in the **centered** metric:

```
coef(b13.3)
```

```
## $dept_id
## , , Intercept
##
##      Estimate   Est.Error    Q2.5    Q97.5
## 1  1.3000810 0.25800931 0.7956920 1.8180656
## 2  0.7427406 0.32186659 0.1135683 1.4015197
## 3 -0.6471537 0.08493136 -0.8165215 -0.4817777
## 4 -0.6181862 0.10557801 -0.8258275 -0.4116560
## 5 -1.1308586 0.11440283 -1.3611516 -0.9106359
## 6 -2.6039294 0.20068854 -3.0073852 -2.2276502
##
## , , male
##
##      Estimate   Est.Error    Q2.5    Q97.5
## 1 -0.78709583 0.2713916 -1.3260723 -0.2557420
## 2 -0.21265964 0.3240316 -0.8706022 0.4209622
## 3  0.08244543 0.1384462 -0.1838069 0.3577488
## 4 -0.09210853 0.1398596 -0.3661776 0.1831918
## 5  0.12038134 0.1862482 -0.2329588 0.4945237
## 6 -0.12006852 0.2689671 -0.6600662 0.4053155
```

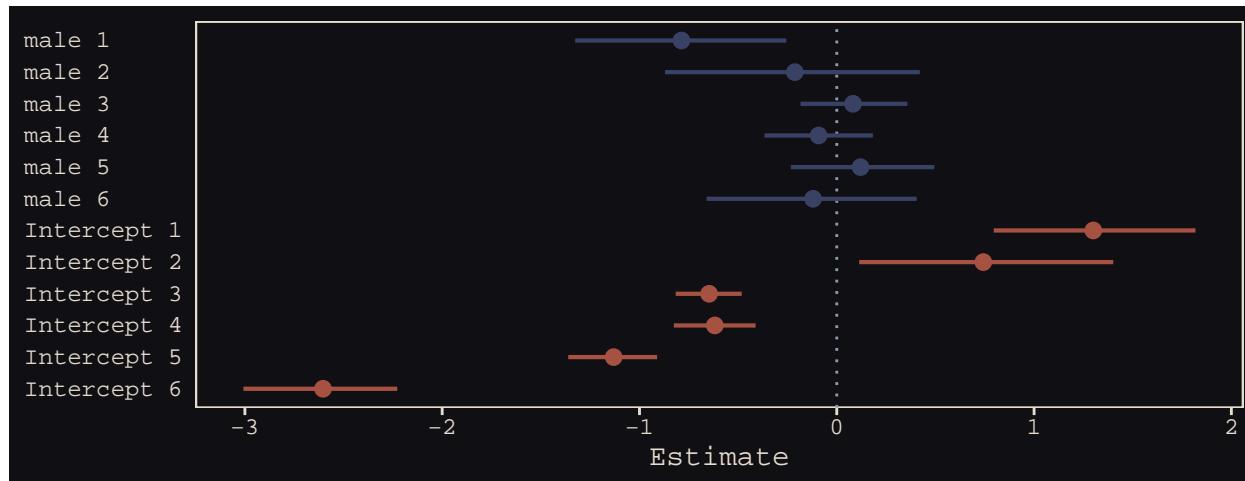
We may as well keep our doing-things-by-hand kick going. Instead relying on `bayesplot::mcmc_intervals()` or `tidybayes::pointintervalh()` to make our coefficient plot, we'll combine `geom_pointrange()` and `coord_flip()`. But we will need to wrangle a bit to get those brms-based **centered** random effects into a usefully-formatted tidy tibble.

```
# as far as I can tell, because `coef()` yields a list, you have to take out the two
# random effects one at a time and then bind them together to get them ready for a tibble
rbind(coef(b13.3)$dept_id[, , 1],
      coef(b13.3)$dept_id[, , 2]) %>%
  as_tibble() %>%
  mutate(param = c(paste("Intercept", 1:6), paste("male", 1:6)),
         reorder = c(6:1, 12:7)) %>%
  # plot
  ggplot(aes(x = reorder(param, reorder))) +
```

```

geom_hline(yintercept = 0, linetype = 3, color = "#8B9DAF") +
geom_pointrange(aes(ymin = Q2.5, ymax = Q97.5, y = Estimate, color = reorder < 7),
                 shape = 20, size = 3/4) +
scale_color_manual(values = c("#394165", "#A65141")) +
xlab(NULL) +
coord_flip() +
theme_pearl_earring +
theme(legend.position = "none",
      axis.ticks.y = element_blank(),
      axis.text.y = element_text(hjust = 0))

```



Just like in the text, our `male` slopes are much less dispersed than our intercepts.

13.2.3 Shrinkage.

Figure 13.6.a depicts the correlation between the full UCB model's varying intercepts and slopes.

```

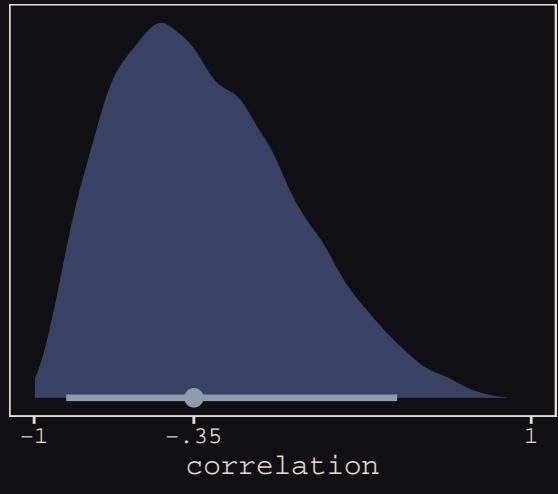
library(tidybayes)

post <- posterior_samples(b13.3)

post %>%
  ggplot(aes(x = cor_dept_id__Intercept__male, y = 0)) +
  geom_halfeyeh(fill = "#394165", color = "#8B9DAF",
                 point_interval = median_qi, .width = .95) +
  scale_x_continuous(breaks = c(-1, median(post$cor_dept_id__Intercept__male), 1),
                     labels = c(-1, "-.35", 1)) +
  scale_y_continuous(NULL, breaks = NULL) +
  coord_cartesian(xlim = -1:1) +
  labs(subtitle = "The dot is at the median; the\nhorizontal bar is the 95% CI.",
       x = "correlation") +
  theme_pearl_earring

```

The dot is at the median; the horizontal bar is the 95% CI.



Much like for Figure 13.5.b, above, it'll take a little data processing before we're ready to reproduce Figure 13.6.b.

```
# here we put the partially-pooled estimate summaries in a tibble
partially_pooled_params <-
  coef(b13.3)$dept_id[, 1, ] %>%
  as_tibble() %>%
  set_names("intercept", "slope") %>%
  mutate(dept = 1:n()) %>%
  select(dept, everything())

# in order to calculate the unpooled estimates from the data, we'll need a function that
# can convert probabilities into the logit metric. if you do the algebra, this is just
# a transformation of the `inv_logit_scaled()` function.
prob_to_logit <- function(x){
  -log((1 / x) - 1)
}

# compute unpooled estimates directly from data
un_pooled_params <-
  d %>%
  group_by(male, dept_id) %>%
  summarise(prob_admit = mean(admit / applications)) %>%
  ungroup() %>%
  mutate(male = ifelse(male == 0, "intercept", "slope")) %>%
  spread(key = male, value = prob_admit) %>%
  rename(dept = dept_id) %>%
  # here we put our `prob_to_logit()` function to work
  mutate(intercept = prob_to_logit(intercept),
         slope      = prob_to_logit(slope)) %>%
  mutate(slope      = slope - intercept)

# here we combine the partially-pooled and unpooled means into a single data object
params <-
  bind_rows(partially_pooled_params, un_pooled_params) %>%
  mutate(pooled      = rep(c("partially", "not"), each = n() / 2)) %>%
  mutate(dept_letter = rep(LETTERS[1:6], times = 2)) # this will help with plotting

params

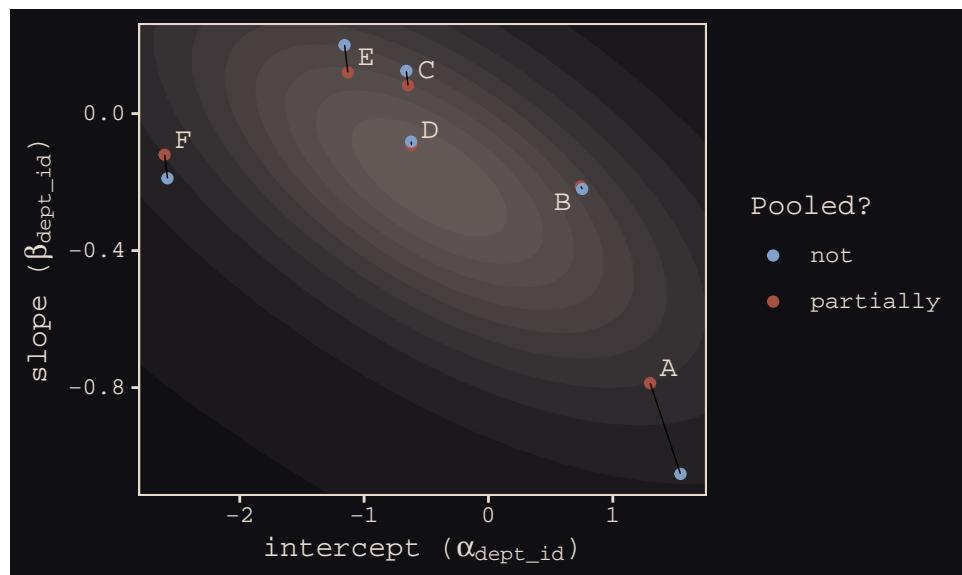
## # A tibble: 12 x 5
##       dept intercept     slope pooled dept_letter
## *   <dbl>     <dbl>     <dbl> <chr>    <chr>
## 1    -0.35     -0.35     -0.35 partially A
## 2    -0.35     -0.35     -0.35 partially B
## 3    -0.35     -0.35     -0.35 partially C
## 4    -0.35     -0.35     -0.35 partially D
## 5    -0.35     -0.35     -0.35 partially E
## 6    -0.35     -0.35     -0.35 partially F
## 7     0.35      0.35      0.35    not    A
## 8     0.35      0.35      0.35    not    B
## 9     0.35      0.35      0.35    not    C
## 10    0.35      0.35      0.35    not    D
## 11    0.35      0.35      0.35    not    E
## 12    0.35      0.35      0.35    not    F
```

```
##   <int>   <dbl>   <dbl> <chr>      <chr>
## 1     1    1.30 -0.787 partially A
## 2     2    0.743 -0.213 partially B
## 3     3   -0.647  0.0824 partially C
## 4     4   -0.618 -0.0921 partially D
## 5     5   -1.13   0.120 partially E
## 6     6   -2.60   -0.120 partially F
## 7     1    1.54  -1.05  not       A
## 8     2    0.754 -0.220  not       B
## 9     3   -0.660  0.125  not       C
## 10    4   -0.622 -0.0820 not       D
## 11    5   -1.16   0.200  not       E
## 12    6   -2.58  -0.189  not       F
```

Here's our version of Figure 13.6.b, depicting two-dimensional shrinkage for the partially-pooled multilevel estimates (posterior means) relative to the unpooled coefficients, calculated from the data. The `ggrepel::geom_text_repel()` function will help us with the in-plot labels.

```
library(ggrepel)

ggplot(data = params, aes(x = intercept, y = slope)) +
  mapply(function(level){
    stat_ellipse(geom = "polygon", type = "norm",
                 size = 0, alpha = 1/20, fill = "#E7CDC2",
                 level = level)
  },
  level = c(seq(from = 1/10, to = 9/10, by = 1/10), .99)) +
  geom_point(aes(group = dept, color = pooled)) +
  geom_line(aes(group = dept), size = 1/4) +
  scale_color_manual("Pooled?",
                     values = c("#80A0C7", "#A65141")) +
  geom_text_repel(data = params %>% filter(pooled == "partially"),
                 aes(label = dept_letter),
                 color = "#E8DCCF", size = 4, family = "Courier", seed = 13.6) +
  coord_cartesian(xlim = range(params$intercept),
                  ylim = range(params$slope)) +
  labs(x = expression(paste("intercept (", alpha[dept_id], ")")),
       y = expression(paste("slope (", beta[dept_id], ")"))) +
  theme_pearl_earring
```



13.2.4 Model comparison.

Fit the no-gender model.

```
b13.4 <-
  brm(data = d, family = binomial,
    admit | trials(applications) ~ 1 + (1 | dept_id),
    prior = c(prior(normal(0, 10), class = Intercept),
              prior(cauchy(0, 2), class = sd)),
    iter = 5000, warmup = 1000, chains = 4, cores = 4,
    seed = 13,
    control = list(adapt_delta = .99,
                   max_treedepth = 12))
```

Compare the three models by the WAIC.

```
b13.2 <- add_criterion(b13.2, "waic")
b13.3 <- add_criterion(b13.3, "waic")
b13.4 <- add_criterion(b13.4, "waic")

loo_compare(b13.2, b13.3, b13.4, criterion = "waic") %>%
  print(simplify = F)

##      elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic se_waic
## b13.3    0.0     0.0   -45.6      2.4       6.9     1.5    91.2    4.9
## b13.4   -7.0     7.3   -52.6      9.0       6.5     2.3   105.1   18.0
## b13.2   -8.5     6.4   -54.1      8.1       9.2     2.9   108.2   16.3
```

In terms of the WAIC estimates and elpd differences, the models are similar. The story changes when we look at the WAIC weights.

```
model_weights(b13.2, b13.3, b13.4, weights = "waic") %>%
  round(digits = 3)

## b13.2 b13.3 b13.4
## 0.000 0.999 0.001
```

The varying slopes model, [b13.3], dominates [the other two]. This is despite the fact that the *average* slope in [b13.3] is nearly zero. The average isn't what matters, however. It is the individual slopes, one for each department, that matter. If we wish to generalize to new departments, the variation in slopes suggest that it'll be worth paying attention to gender, even if the average slope is nearly zero in the population. (pp. 402–403, *emphasis* in the original)

13.3 Example: Cross-classified chimpanzees with varying slopes

Retrieve the chimpanzees data.

```
library(rethinking)
data(chimpanzees)
d <- chimpanzees

detach(package:rethinking, unload = T)
library(brms)
rm(chimpanzees)
```

```
d <-
  d %>%
  select(-recipient) %>%
  mutate(block_id = block)
```

My math's aren't the best. But if I'm following along correctly, here's a fuller statistical expression of our cross-classified model.

$$\begin{aligned}
 \text{pulled_left}_i &\sim \text{Binomial}(n = 1, p_i) \\
 \text{logit}(p_i) &= \alpha_i + (\beta_{1i} + \beta_{2i}\text{condition}_i)\text{prosoc_left}_i \\
 \alpha_i &= \alpha + \alpha_{\text{actor}_i} + \alpha_{\text{block_id}_i} \\
 \beta_{1i} &= \beta_1 + \beta_{1,\text{actor}_i} + \beta_{1,\text{block_id}_i} \\
 \beta_{2i} &= \beta_2 + \beta_{2,\text{actor}_i} + \beta_{2,\text{block_id}_i} \\
 \begin{bmatrix} \alpha_{\text{actor}} \\ \beta_{1,\text{actor}} \\ \beta_{2,\text{actor}} \end{bmatrix} &\sim \text{MVNormal} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \mathbf{S}_{\text{actor}} \right) \\
 \begin{bmatrix} \alpha_{\text{block_id}} \\ \beta_{1,\text{block_id}} \\ \beta_{2,\text{block_id}} \end{bmatrix} &\sim \text{MVNormal} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \mathbf{S}_{\text{block_id}} \right) \\
 \mathbf{S}_{\text{actor}} &= \begin{pmatrix} \sigma_{\alpha_{\text{actor}}} & 0 & 0 \\ 0 & \sigma_{\beta_{1,\text{actor}}} & 0 \\ 0 & 0 & \sigma_{\beta_{2,\text{actor}}} \end{pmatrix} \mathbf{R}_{\text{actor}} \begin{pmatrix} \sigma_{\alpha_{\text{actor}}} & 0 & 0 \\ 0 & \sigma_{\beta_{1,\text{actor}}} & 0 \\ 0 & 0 & \sigma_{\beta_{2,\text{actor}}} \end{pmatrix} \\
 \mathbf{S}_{\text{block_id}} &= \begin{pmatrix} \sigma_{\alpha_{\text{block_id}}} & 0 & 0 \\ 0 & \sigma_{\beta_{1,\text{block_id}}} & 0 \\ 0 & 0 & \sigma_{\beta_{2,\text{block_id}}} \end{pmatrix} \mathbf{R}_{\text{block_id}} \begin{pmatrix} \sigma_{\alpha_{\text{block_id}}} & 0 & 0 \\ 0 & \sigma_{\beta_{1,\text{block_id}}} & 0 \\ 0 & 0 & \sigma_{\beta_{2,\text{block_id}}} \end{pmatrix} \\
 \alpha &\sim \text{Normal}(0, 1) \\
 \beta_1 &\sim \text{Normal}(0, 1) \\
 \beta_2 &\sim \text{Normal}(0, 1) \\
 (\sigma_{\alpha_{\text{actor}}}, \sigma_{\beta_{1,\text{actor}}}, \sigma_{\beta_{2,\text{actor}}}) &\sim \text{HalfCauchy}(0, 2) \\
 (\sigma_{\alpha_{\text{block_id}}}, \sigma_{\beta_{1,\text{block_id}}}, \sigma_{\beta_{2,\text{block_id}}}) &\sim \text{HalfCauchy}(0, 2) \\
 \mathbf{R}_{\text{actor}} &\sim \text{LKJcorr}(4) \\
 \mathbf{R}_{\text{block_id}} &\sim \text{LKJcorr}(4)
 \end{aligned}$$

And now each \mathbf{R} is a 3×3 correlation matrix.

Let's fit this beast.

```
b13.6 <-
  brm(data = d, family = binomial,
    pulled_left | trials(1) ~ 1 + prosoc_left + condition:prosoc_left +
      (1 + prosoc_left + condition:prosoc_left | actor) +
      (1 + prosoc_left + condition:prosoc_left | block_id),
    prior = c(prior(normal(0, 1), class = Intercept),
              prior(normal(0, 1), class = b),
              prior(cauchy(0, 2), class = sd),
              prior(lkj(4), class = cor)),
    iter = 5000, warmup = 1000, chains = 3, cores = 3,
    seed = 13)
```

Even though it's not apparent in the syntax, our model b13.6 was already fit using the [non-centered parameterization](#). Behind the scenes, Bürkner has `brms` do this automatically. It's been that way all along.

If you recall from last chapter, we can compute the number of effective samples for our parameters like so.

```

ratios_cp <- neff_ratio(b13.6)

neff <-
  ratios_cp %>%
  as_tibble %>%
  rename(neff_ratio = value) %>%
  mutate(neff      = neff_ratio * 12000)

head(neff)

## # A tibble: 6 x 2
##   neff_ratio    neff
##       <dbl>   <dbl>
## 1     0.271 3252.
## 2     0.505 6058.
## 3     0.622 7459.
## 4     0.317 3809.
## 5     0.516 6193.
## 6     0.513 6154.

```

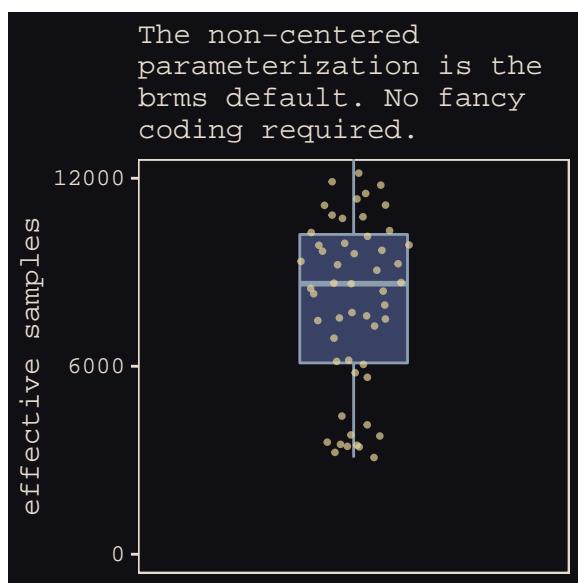
Now we're ready for our variant of Figure 13.7. The handy [ggbeeswarm package](#) and its `geom_quasirandom()` function will give a better sense of the distribution.

```

library(ggbeeswarm)

neff %>%
  ggplot(aes(x = factor(0), y = neff)) +
  geom_boxplot(fill = "#394165", color = "#8B9DAF") +
  geom_quasirandom(method = "tukeyDense",
                    size = 2/3, color = "#EEDA9D", alpha = 2/3) +
  scale_x_discrete(NULL, breaks = NULL,
                    expand = c(.75, .75)) +
  scale_y_continuous(breaks = c(0, 6000, 12000)) +
  coord_cartesian(ylim = 0:12000) +
  labs(y = "effective samples",
       subtitle = "The non-centered\nparameterization is the\nbrms default. No fancy\ncoding required.") +
  theme_pearl_earring

```

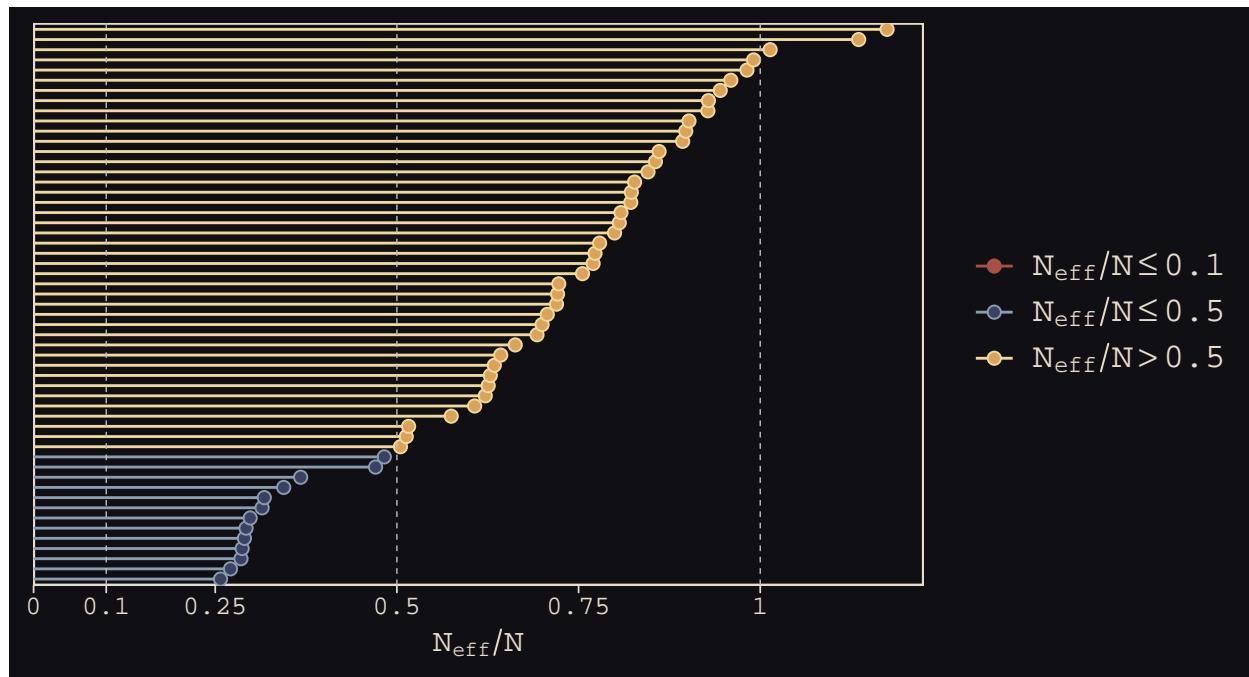


As in the last chapter, we'll use the `bayesplot::mcmc_neff()` function to examine the ratio of `n.eff` and the fill number of post-warm-up iterations, N . Ideally, that ratio is closer to 1 than not.

```
library(bayesplot)

color_scheme_set(c("#DCA258", "#EEDA9D", "#394165", "#8B9DAF", "#A65141", "#A65141"))

mcmc_neff(ratios_cp, size = 2) +
  theme_pearl_earring
```



Here are our standard deviation parameters.

```
tidy(b13.6) %>%
  filter(str_detect(term, "sd_")) %>%
  mutate_if(is.numeric, round, digits = 2)

## #>
## #> #> term estimate std.error lower upper
## #> 1 sd_actor__Intercept 2.31 0.89 1.28 3.91
## #> 2 sd_actor__prosoc_left 0.46 0.37 0.04 1.13
## #> 3 sd_actor__prosoc_left:condition 0.52 0.47 0.04 1.40
## #> 4 sd_block_id__Intercept 0.23 0.20 0.02 0.60
## #> 5 sd_block_id__prosoc_left 0.57 0.40 0.06 1.33
## #> 6 sd_block_id__prosoc_left:condition 0.53 0.44 0.04 1.34
```

McElreath discussed `rethinking::link()` in the middle of page 407. He showed how his `link(m13.6NC)` code returned a list of four matrices, of which the `p` matrix was of primary interest. The `brms::fitted()` function doesn't work quite the same way, here.

```
fitted(b13.6,
       summary = F,
       nsamples = 1000) %>%
str()

## #> num [1:1000, 1:504] 0.47 0.323 0.306 0.459 0.396 ...
```

First off, recall that `fitted()` returns summary values, by default. If we want individual values, set `summary = FALSE`. It's also the `fitted()` default to use all posterior iterations, which is 12,000 in this case. To match the text, we need to set `nsamples = 1000`. But those are just details. The main point is that `fitted()` only returns one matrix, which is the analogue to the `p` matrix in the text.

Moving forward, before we can follow along with McElreath's R code 13.27, we need to refit the simpler model from way back in Chapter 12.

```
b12.5 <-
  brm(data = d, family = binomial,
       pulled_left | trials(1) ~ 1 + prosoc_left + condition:prosoc_left +
         (1 | actor) + (1 | block_id),
       prior = c(prior(normal(0, 10), class = Intercept),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 1), class = sd)),
       iter = 5000, warmup = 1000, chains = 3, cores = 3,
       seed = 13)
```

Now we can compare them by the WAIC.

```
b13.6 <- add_criterion(b13.6, "waic")
b12.5 <- add_criterion(b12.5, "waic")

loo_compare(b13.6, b12.5, criterion = "waic") %>%
  print(simplify = F)

##          elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic    se_waic
## b12.5      0.0      0.0   -266.4      9.8     10.5     0.5    532.8    19.7
## b13.6     -1.0      2.0   -267.4      9.9     18.4     0.9    534.9    19.9

model_weights(b13.6, b12.5, weights = "waic")

##      b13.6      b12.5
## 0.2635335 0.7364665
```

In this example, no matter which varying effect structure you use, you'll find that actors vary a lot in their baseline preference for the left-hand lever. Everything else is much less important. But using the most complex model, [b13.6], tells the correct story. Because the varying slopes are adaptively regularized, the model hasn't overfit much, relative to the simpler model that contains only the important intercept variation. (p. 408)

13.4 Continuous categories and the Gaussian process

There is a way to apply the varying effects approach to continuous categories... The general approach is known as Gaussian process regression. This name is unfortunately wholly uninformative about what it is for and how it works.

We'll proceed to work through a basic example that demonstrates both what it is for and how it works. The general purpose is to define some dimension along which cases differ. This might be individual differences in age. Or it could be differences in location. Then we measure the distance between each pair of cases. What the model then does is estimate a function for the covariance between pairs of cases at different distances. This covariance function provides one continuous category generalization of the varying effects approach. (p. 410)

13.4.1 Example: Spatial autocorrelation in Oceanic tools.

```
# load the distance matrix
library(rethinking)
data(islandsDistMatrix)

# display short column names, so fits on screen
d_mat <- islandsDistMatrix
colnames(d_mat) <- c("M1", "Ti", "SC", "Ya", "Fi",
                     "Tr", "Ch", "Mn", "To", "Ha")
round(d_mat, 1)
```

```
##          Ml  Ti  SC  Ya  Fi  Tr  Ch  Mn  To  Ha
## Malekula  0.0 0.5 0.6 4.4 1.2 2.0 3.2 2.8 1.9 5.7
## Tikopia   0.5 0.0 0.3 4.2 1.2 2.0 2.9 2.7 2.0 5.3
## Santa Cruz 0.6 0.3 0.0 3.9 1.6 1.7 2.6 2.4 2.3 5.4
## Yap       4.4 4.2 3.9 0.0 5.4 2.5 1.6 1.6 6.1 7.2
## Lau Fiji  1.2 1.2 1.6 5.4 0.0 3.2 4.0 3.9 0.8 4.9
## Trobriand 2.0 2.0 1.7 2.5 3.2 0.0 1.8 0.8 3.9 6.7
## Chuuk     3.2 2.9 2.6 1.6 4.0 1.8 0.0 1.2 4.8 5.8
## Manus     2.8 2.7 2.4 1.6 3.9 0.8 1.2 0.0 4.6 6.7
## Tonga     1.9 2.0 2.3 6.1 0.8 3.9 4.8 4.6 0.0 5.0
## Hawaii    5.7 5.3 5.4 7.2 4.9 6.7 5.8 6.7 5.0 0.0
```

If you wanted to use color to more effectively visualize the values in the matrix, you might do something like this.

```
d_mat %>%
  as_tibble() %>%
  gather() %>%
  rename(column = key,
         distance = value) %>%
  mutate(row      = rep(rownames(d_mat), times = 10),
         row_order = rep(9:0,           times = 10),
         column_order = rep(0:9,        each = 10)) %>%

  ggplot(aes(x = reorder(column, column_order),
             y = reorder(row,   row_order))) +
  geom_raster(aes(fill = distance)) +
  geom_text(aes(label = round(distance, digits = 1)),
            size = 3, family = "Courier", color = "#100F14") +
  scale_fill_gradient(low = "#FCF9F0", high = "#A65141") +
  scale_x_discrete(NULL, position = "top", expand = c(0, 0)) +
  scale_y_discrete(NULL, expand = c(0, 0)) +
  theme_pearl_earring +
  theme(axis.ticks = element_blank(),
        axis.text.y = element_text(hjust = 0))
```

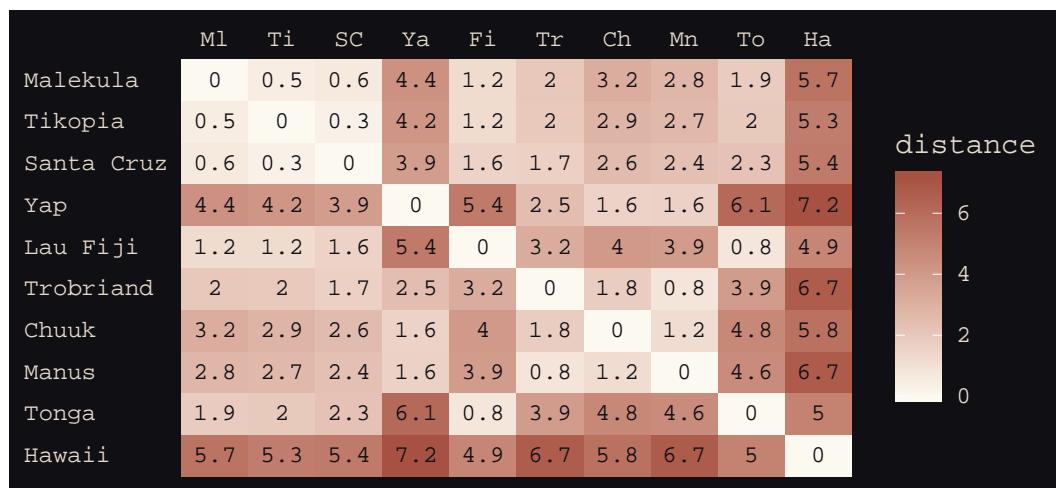


Figure 13.8 shows the “shape of the function relating distance to the covariance \mathbf{K}_{ij} .”

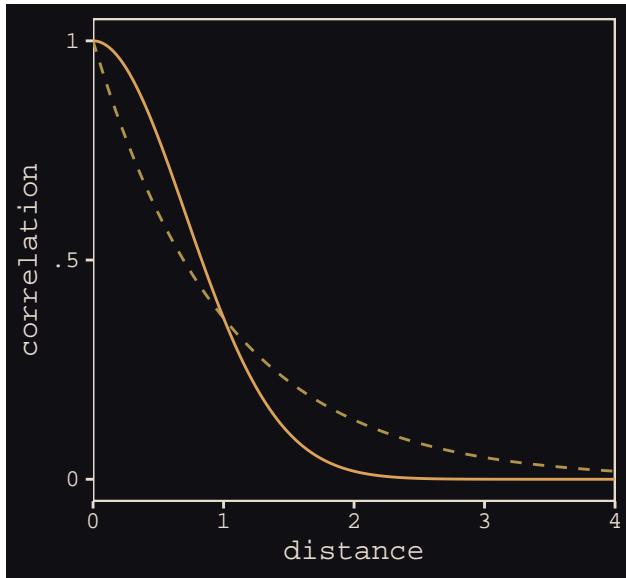
```
tibble(
  x      = seq(from = 0, to = 4, by = .01),
  linear = exp(-1 * x),
  squared = exp(-1 * x^2)) %>%

  ggplot(aes(x = x)) +
  geom_line(aes(y = linear),
```

```

      color = "#B1934A", linetype = 2) +
geom_line(aes(y = squared),
          color = "#DCA258") +
scale_x_continuous("distance", expand = c(0, 0)) +
scale_y_continuous("correlation",
                   breaks = c(0, .5, 1),
                   labels = c(0, ".5", 1)) +
theme_pearl_earring

```



Load the data.

```

data(Kline2) # load the ordinary data, now with coordinates

d <-
  Kline2 %>%
  mutate(society = 1:10)

rm(Kline2)

d %>% glimpse()

## Observations: 10
## Variables: 10
## $ culture      <fct> Malekula, Tikopia, Santa Cruz, Yap, Lau Fiji, Trobriand, Chuuk, Manus, Tong...
## $ population   <int> 1100, 1500, 3600, 4791, 7400, 8000, 9200, 13000, 17500, 275000
## $ contact      <fct> low, low, low, high, high, high, high, low, high, low
## $ total_tools  <int> 13, 22, 24, 43, 33, 19, 40, 28, 55, 71
## $ mean_TU      <dbl> 3.2, 4.7, 4.0, 5.0, 5.0, 4.0, 3.8, 6.6, 5.4, 6.6
## $ lat           <dbl> -16.3, -12.3, -10.7, 9.5, -17.7, -8.7, 7.4, -2.1, -21.2, 19.9
## $ lon           <dbl> 167.5, 168.8, 166.0, 138.1, 178.1, 150.9, 151.6, 146.9, -175.2, -155.6
## $ lon2          <dbl> -12.5, -11.2, -14.0, -41.9, -1.9, -29.1, -28.4, -33.1, 4.8, 24.4
## $ logpop        <dbl> 7.003065, 7.313220, 8.188689, 8.474494, 8.909235, 8.987197, 9.126959, 9.472...
## $ society       <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```

Switch out rethinking for brms.

```

detach(package:rethinking, unload = T)
library(brms)

```

Okay, it appears this is going to be a bit of a ride. It's not entirely clear to me if we can fit a Gaussian process model in brms that's a direct equivalent to what McElreath did with rethinking. But we can try. First, note our use of the `gp()` syntax in the `brm()` function, below. We're attempting to tell brms that we would like to include latitude and longitude (i.e., `lat` and `long2`, respectively) in a Gaussian process. Also note how our priors are a little different than those in the text. I'll explain, below. Let's just move ahead and fit the model.

```
b13.7 <-  
  brm(data = d, family = poisson,  
    total_tools ~ 1 + gp(lat, lon2) + logpop,  
    prior = c(prior(normal(0, 10), class = Intercept),  
              prior(normal(0, 1), class = b),  
              prior(inv_gamma(2.874624, 0.393695), class = lscale),  
              prior(cauchy(0, 1), class = sdgp)),  
    iter = 1e4, warmup = 2000, chains = 4, cores = 4,  
    seed = 13,  
    control = list(adapt_delta = 0.999,  
                  max_treedepth = 12))
```

Here's the model summary.

```
posterior_summary(b13.7) %>%  
  round(digits = 2)
```

	Estimate	Est.Error	Q2.5	Q97.5
## b_Intercept	1.43	1.11	-0.80	3.71
## b_logpop	0.24	0.11	0.02	0.45
## sdgp_gplation2	0.53	0.36	0.16	1.43
## lscale_gplation2	0.23	0.13	0.07	0.56
## zgp_gplation2[1]	-0.58	0.79	-2.14	0.96
## zgp_gplation2[2]	0.44	0.84	-1.26	2.07
## zgp_gplation2[3]	-0.63	0.70	-1.98	0.89
## zgp_gplation2[4]	0.88	0.70	-0.46	2.29
## zgp_gplation2[5]	0.25	0.76	-1.25	1.75
## zgp_gplation2[6]	-0.99	0.80	-2.54	0.62
## zgp_gplation2[7]	0.13	0.73	-1.40	1.55
## zgp_gplation2[8]	-0.18	0.88	-1.91	1.61
## zgp_gplation2[9]	0.41	0.93	-1.51	2.16
## zgp_gplation2[10]	-0.32	0.83	-1.95	1.30
## lp_	-51.53	3.18	-58.66	-46.33

Our Gaussian process parameters are different than McElreath's. From the brms reference manual, here's the brms parameterization:

$$k(x_i, x_j) = sdgp^2 \exp\left(-\|x_i - x_j\|^2 / (2lscale^2)\right)$$

What McElreath called η , Bürkner called `sdgp`. While McElreath estimated η^2 , brms simply estimated `sdgp`. So we'll have to square our `sdgp_gplation2` before it's on the same scale as `etasq` in the text. Here it is.

```
posterior_samples(b13.7) %>%  
  transmute(sdgp_squared = sdgp_gplation2^2) %>%  
  mean_hdi(sdgp_squared, .width = .89) %>%  
  mutate_if(is.double, round, digits = 3)
```

```
##   sdgp_squared .lower .upper .width .point .interval  
## 1          0.402      0  0.77  0.89   mean       hdi
```

Now we're in the ballpark. In our model `brm()` code, above, we just went with the flow and kept the `cauchy(0, 1)` prior on `sdgp`.

Now look at the denominator of the inner part of Bürkner equation, $2l\text{scale}^2$. This appears to be the brms equivalent to McElreath's ρ^2 . Or at least it's what we've got. Anyway, also note that McElreath estimated ρ^2 directly as `rhosq`. If I'm doing the algebra correctly—and that may well be a big if—, we might expect:

$$\rho^2 = 1/(2 \cdot l\text{scale}^2)$$

But that doesn't appear to be the case. *Sigh.*

```
posterior_samples(b13.7) %>%
  transmute(rho_squared = 1 / (2 * lscale_gplatlon2^2)) %>%
  mean_hdi(rho_squared, .width = .89) %>%
  mutate_if(is.double, round, digits = 3)
```

```
##   rho_squared .lower .upper .width .point .interval
## 1      21.354  0.481 46.174    0.89     mean       hdi
```

Oh man, that isn't even close to the 2.67 McElreath reported in the text. The plot deepens. If you look back, you'll see we used a very different prior for `lscale`. Here is it: `inv_gamma(2.874624, 0.393695)`. Use `get_prior()` to discover where that came from.

```
get_prior(data = d, family = poisson,
          total_tools ~ 1 + gp(lat, lon2) + logpop)
```

	prior	class	coef	group	resp	dpar	nlpar	bound
## 1		b						
## 2		b	logpop					
## 3	student_t(3, 3, 10)	Intercept						
## 4	normal(0, 0.5)	lscale						
## 5	inv_gamma(2.874624, 0.393695)	lscale	gplatlon2					
## 6	student_t(3, 0, 10)	sdgp						
## 7		sdgp	gplatlon2					

That is, we used the brms default prior for `lscale`. In a [GitHub exchange](#), Bürkner pointed out that brms uses special priors for `lscale` parameters based on [Michael Betancourt](#) [of the Stan team]'s [vignette on the topic](#). Though it isn't included in this document, I also ran the model with the `cauchy(0, 1)` prior and the results were quite similar. So the big discrepancy between our model and the one in the text isn't based on that prior.

Now that we've hopped on the comparison train, we may as well keep going down the track. Let's reproduce McElreath's model with rethinking.

Switch out brms for rethinking.

```
detach(package:brms, unload = T)
library(rethinking)
```

Now fit the `rethinking::map2stan()` model.

```
m13.7 <- map2stan(
  alist(
    total_tools ~ dpois(lambda),
    log(lambda) <- a + g[society] + bp*logpop,
    g[society] ~ GPL2( Dmat , etasq , rhosq , 0.01 ),
    a ~ dnorm(0,10),
    bp ~ dnorm(0,1),
    etasq ~ dcauchy(0,1),
    rhosq ~ dcauchy(0,1)
  ),
  data=list(
```

```
total_tools=d$total_tools,
logpop=d$logpop,
society=d$society,
Dmat=islandsDistMatrix),
warmup=2000 , iter=1e4 , chains=4)
```

Alright, now we'll work directly with the posteriors to make some visual comparisons.

```
# rethinking-based posterior
post_m13.7 <- rethinking::extract.samples(m13.7)[2:5] %>% as_tibble()

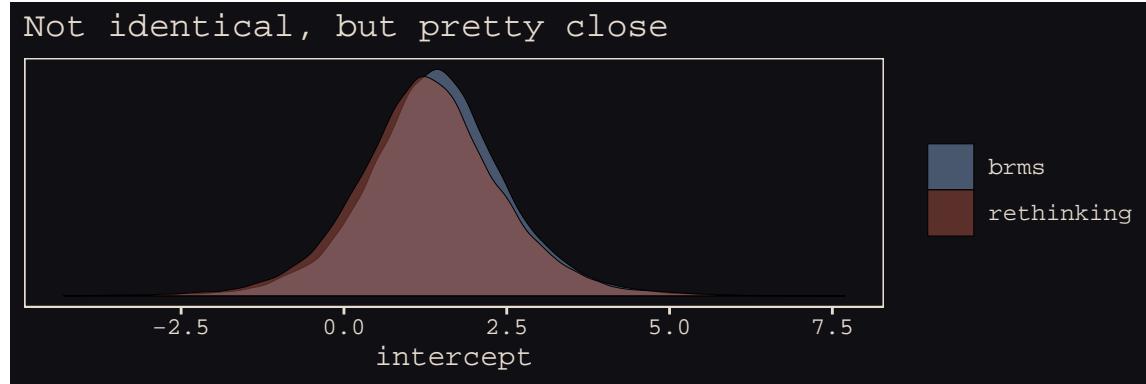
detach(package:rethinking, unload = T)
library(brms)

# brms-based posterior
post_b13.7 <- posterior_samples(b13.7)
```

Here's the model intercept posterior, by package.

```
post_m13.7[, "a"] %>%
bind_rows(post_b13.7 %>%
          transmute(a = b_Intercept)) %>%
mutate(package = rep(c("rethinking", "brms"), each = nrow(post_m13.7))) %>%

ggplot(aes(x = a, fill = package)) +
geom_density(size = 0, alpha = 1/2) +
scale_fill_manual(NULL, values = c("#80A0C7", "#A65141")) +
scale_y_continuous(NULL, breaks = NULL) +
labs(title = "Not identical, but pretty close",
     x      = "intercept") +
theme_pearl_earring
```

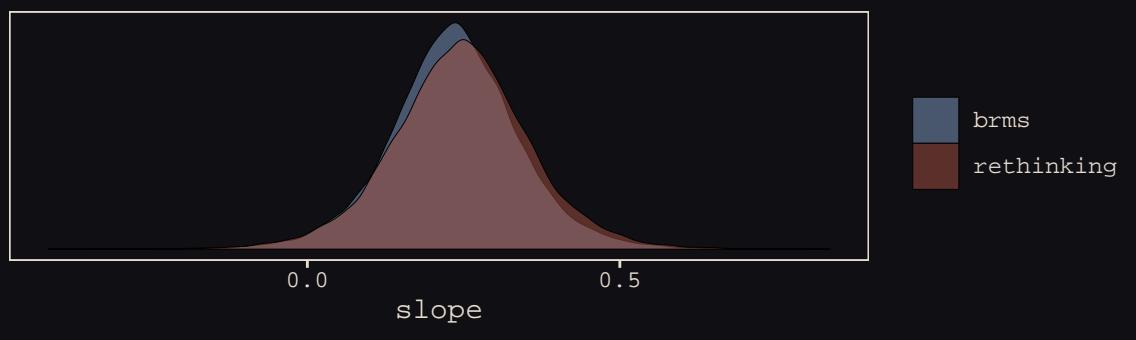


Now check the slopes.

```
post_m13.7[, "bp"] %>%
bind_rows(post_b13.7 %>%
          transmute(bp = b_logpop)
          ) %>%
mutate(package = rep(c("rethinking", "brms"), each = nrow(post_m13.7))) %>%

ggplot(aes(x = bp, fill = package)) +
geom_density(size = 0, alpha = 1/2) +
scale_fill_manual(NULL, values = c("#80A0C7", "#A65141")) +
scale_y_continuous(NULL, breaks = NULL) +
labs(title = "Again, pretty close",
     x      = "slope") +
theme_pearl_earring
```

Again, pretty close

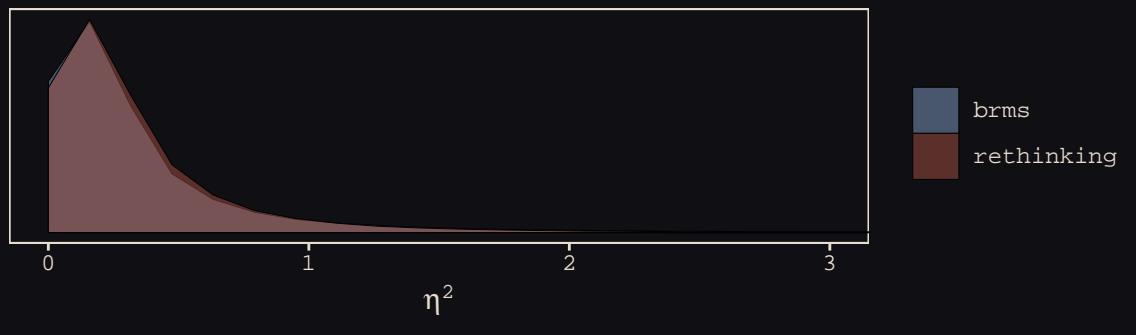


This one, η^2 , required a little transformation.

```
post_m13.7[, "etasq"] %>%
  bind_rows(post_b13.7 %>%
    transmute(etasq = sdgp_gplation2^2)) %>%
  mutate(package = rep(c("rethinking", "brms"), each = nrow(post_m13.7))) %>%

  ggplot(aes(x = etasq, fill = package)) +
  geom_density(size = 0, alpha = 1/2) +
  scale_fill_manual(NULL, values = c("#80A0C7", "#A65141")) +
  scale_y_continuous(NULL, breaks = NULL) +
  labs(title = "Still in the same ballpark",
       x      = expression(eta^2)) +
  coord_cartesian(xlim = 0:3) +
  theme_pearl_earring
```

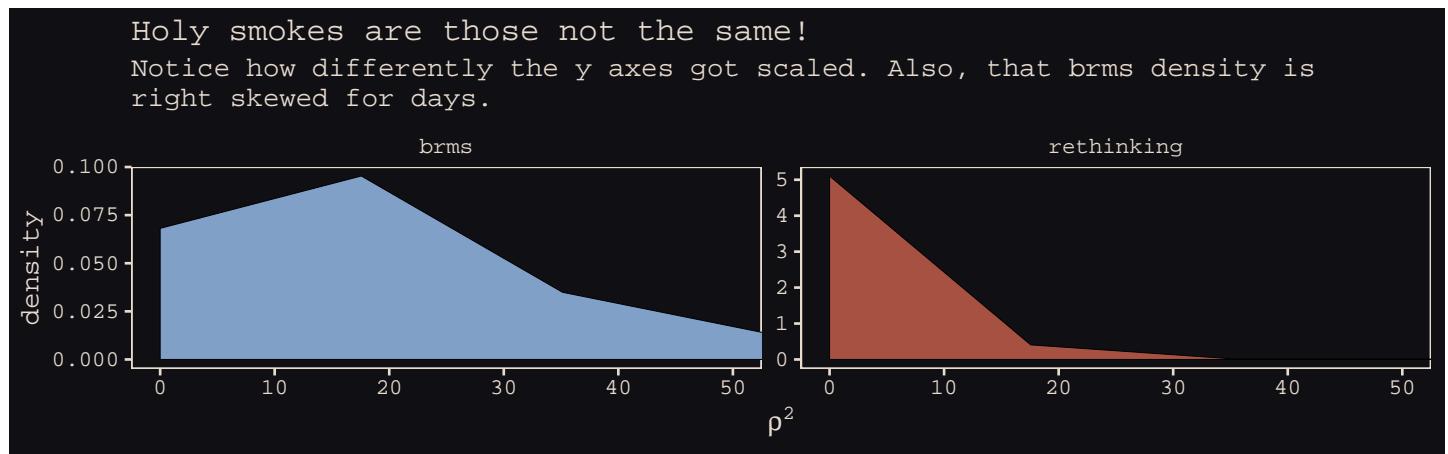
Still in the same ballpark



ρ^2 required more extensive transformation of the brms posterior:

```
post_m13.7[, "rhosq"] %>%
  bind_rows(post_b13.7 %>%
    transmute(rhosq = 1 / (2 * (lscale_gplation2^2))) %>%
  mutate(package = rep(c("rethinking", "brms"), each = nrow(post_m13.7))) %>%

  ggplot(aes(x = rhosq, fill = package)) +
  geom_density(size = 0) +
  scale_fill_manual(NULL, values = c("#80A0C7", "#A65141")) +
  labs(title    = "Holy smokes are those not the same!",
       subtitle = "Notice how differently the y axes got scaled. Also, that brms density is\nright skewed for \u03c1\u00b2") +
  x      = expression(rho^2)) +
  coord_cartesian(xlim = 0:50) +
  theme_pearl_earring +
  theme(legend.position = "none") +
  facet_wrap(~package, scales = "free_y")
```



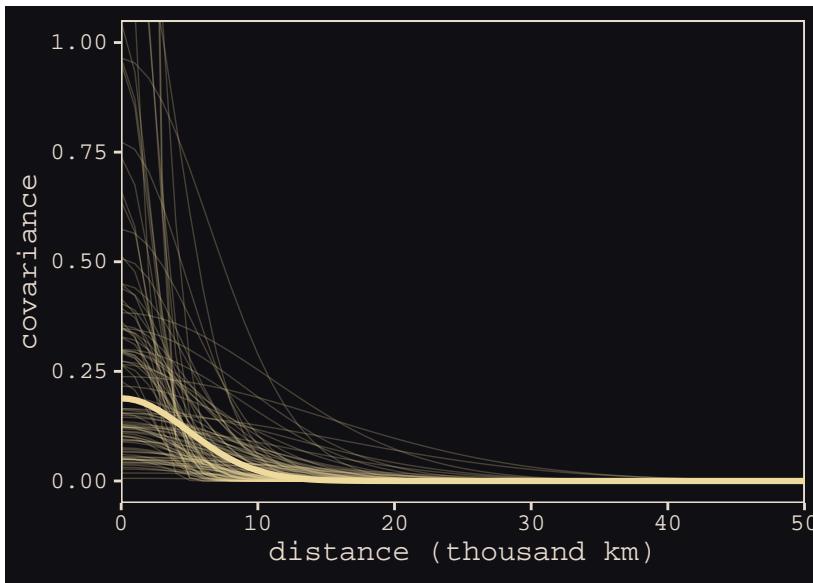
I'm in clinical psychology. Folks in my field don't tend to use Gaussian processes, so getting to the bottom of this is low on my to-do list. Perhaps one of y'all are more experienced with Gaussian processes and see a flaw somewhere in my code. Please [hit me up](#) if you do.

Anyways, here's our brms + ggplot2 version of Figure 13.9.

```
# for `sample_n()`
set.seed(13)

# wrangle
post_b13.7 %>%
  transmute(iter  = 1:n(),
            etasq = sdgp_gplation2^2,
            rhosq = lscale_gplation2^2 * .5) %>%
  sample_n(100) %>%
  expand(nesting(iter, etasq, rhosq),
         x = seq(from = 0, to = 55, by = 1)) %>%
  mutate(covariance = etasq * exp(-rhosq * x^2)) %>%

# plot
ggplot(aes(x = x, y = covariance)) +
  geom_line(aes(group = iter),
            size = 1/4, alpha = 1/4, color = "#EEDA9D") +
  stat_function(fun = function(x) median(post_b13.7$sdgp_gplation2)^2 *
                exp(-median(post_b13.7$lscale_gplation2)^2 *.5 * x^2),
                color = "#EEDA9D", size = 1.1) +
  scale_x_continuous("distance (thousand km)", expand = c(0, 0),
                     breaks = seq(from = 0, to = 50, by = 10)) +
  coord_cartesian(xlim = 0:50,
                  ylim = 0:1) +
  theme_pearl_earring
```



Do note the scale on which we placed our x axis. Our brms parameterization resulted in a gentler decline in spatial covariance. Let's finish this up and "push the parameters back through the function for \mathbf{K} , the covariance matrix" (p. 415).

```
# compute posterior median covariance among societies
k <- matrix(0, nrow = 10, ncol = 10)
for (i in 1:10)
  for (j in 1:10)
    k[i, j] <- median(post_b13.7$sdgp_gplation2^2) *
      exp(-median(post_b13.7$lscale_gplation2^2) *
        islandsDistMatrix[i, j]^2)

diag(k) <- median(post_b13.7$sdgp_gplation2^2) + 0.01

k %>% round(2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,] 0.20 0.19 0.19 0.09 0.18 0.16 0.12 0.14 0.16 0.05
## [2,] 0.19 0.20 0.19 0.09 0.18 0.16 0.13 0.14 0.16 0.06
## [3,] 0.19 0.19 0.20 0.10 0.17 0.17 0.14 0.15 0.15 0.06
## [4,] 0.09 0.09 0.10 0.20 0.06 0.15 0.17 0.17 0.04 0.02
## [5,] 0.18 0.18 0.17 0.06 0.20 0.12 0.10 0.10 0.18 0.07
## [6,] 0.16 0.16 0.17 0.15 0.12 0.20 0.16 0.18 0.10 0.03
## [7,] 0.12 0.13 0.14 0.17 0.10 0.16 0.20 0.18 0.07 0.05
## [8,] 0.14 0.14 0.15 0.17 0.10 0.18 0.18 0.20 0.08 0.03
## [9,] 0.16 0.16 0.15 0.04 0.18 0.10 0.07 0.08 0.20 0.07
## [10,] 0.05 0.06 0.06 0.02 0.07 0.03 0.05 0.03 0.07 0.20
```

And we'll continue to follow suit and change these to a correlation matrix.

```
# convert to correlation matrix
rho <- round(cov2cor(k), 2)

# add row/col names for convenience
colnames(rho) <- c("M1", "Ti", "SC", "Ya", "Fi", "Tr", "Ch", "Mn", "To", "Ha")
rownames(rho) <- colnames(rho)

rho %>% round(2)

##      M1   Ti   SC   Ya   Fi   Tr   Ch   Mn   To   Ha
## M1 1.00 0.94 0.93 0.43 0.89 0.80 0.62 0.69 0.82 0.25
```

```
## Ti 0.94 1.00 0.95 0.46 0.89 0.80 0.67 0.71 0.81 0.30
## SC 0.93 0.95 1.00 0.51 0.86 0.84 0.72 0.75 0.77 0.28
## Ya 0.43 0.46 0.51 1.00 0.28 0.74 0.86 0.85 0.20 0.11
## Fi 0.89 0.89 0.86 0.28 1.00 0.62 0.48 0.50 0.93 0.35
## Tr 0.80 0.80 0.84 0.74 0.62 1.00 0.83 0.92 0.51 0.15
## Ch 0.62 0.67 0.72 0.86 0.48 0.83 1.00 0.89 0.37 0.24
## Mn 0.69 0.71 0.75 0.85 0.50 0.92 0.89 1.00 0.39 0.15
## To 0.82 0.81 0.77 0.20 0.93 0.51 0.37 0.39 1.00 0.33
## Ha 0.25 0.30 0.28 0.11 0.35 0.15 0.24 0.15 0.33 1.00
```

The correlations in our `rho` matrix look a little higher than those in the text. Before we get see them in a plot, let's consider `psize`. If you really want to scale the points in Figure 13.10.a like McElreath did, you can make the `psize` variable in a tidyverse sort of way as follows. However, if you compare the `psize` method and the default `ggplot2` method using just `logpop`, you'll see the difference is negligible. In that light, I'm going to be lazy and just use `logpop` in my plots.

```
d %>%
  transmute(psize = logpop / max(logpop)) %>%
  transmute(psize = exp(psize * 1.5) - 2)
```

```
##          psize
## 1  0.3134090
## 2  0.4009582
## 3  0.6663711
## 4  0.7592196
## 5  0.9066890
## 6  0.9339560
## 7  0.9834797
## 8  1.1096138
## 9  1.2223112
## 10 2.4816891
```

As far as I can figure, you still have to get `rho` into a tidy data frame before feeding it into `ggplot2`. Here's my attempt at doing so.

```
tidy_rho <-
  rho %>%
  data.frame() %>%
  rownames_to_column() %>%
  bind_cols(d %>% select(culture, logpop, total_tools, lon2, lat)) %>%
  gather(colname, correlation, -rowname, -culture, -logpop, -total_tools, -lon2, -lat) %>%
  mutate(group = str_c(pmin(rowname, colname), pmax(rowname, colname))) %>%
  select(rowname, colname, group, culture, everything())

head(tidy_rho)
```

	rowname	colname	group	culture	logpop	total_tools	lon2	lat	correlation	
## 1	Ml	Ml	MlMl	Malekula	7.003065		13	-12.5	-16.3	1.00
## 2	Ti	Ml	MlTi	Tikopia	7.313220		22	-11.2	-12.3	0.94
## 3	SC	Ml	MlSC	Santa Cruz	8.188689		24	-14.0	-10.7	0.93
## 4	Ya	Ml	MlYa	Yap	8.474494		43	-41.9	9.5	0.43
## 5	Fi	Ml	FiMl	Lau Fiji	8.909235		33	-1.9	-17.7	0.89
## 6	Tr	Ml	MlTr	Trobriand	8.987197		19	-29.1	-8.7	0.80

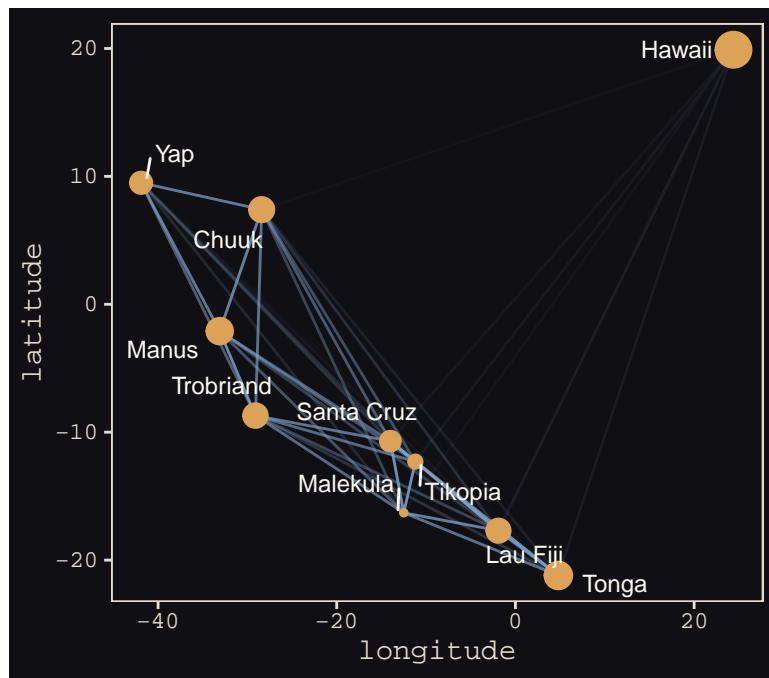
Okay, here's our version of Figure 13.10.a.

```
tidy_rho %>%
  ggplot(aes(x = lon2, y = lat)) +
  geom_line(aes(group = group, alpha = correlation^2),
```

```

    color = "#80A0C7") +
geom_point(data = d, aes(size = logpop), color = "#DCA258") +
geom_text_repel(data = d, aes(label = culture),
seed = 0, point.padding = .3, size = 3, color = "#FCF9F0") +
scale_alpha_continuous(range = c(0, 1)) +
labs(x = "longitude",
y = "latitude") +
coord_cartesian(xlim = range(d$lon2),
ylim = range(d$lat)) +
theme(legend.position = "none") +
theme_pearl_earring

```



Yep, as expressed by the intensity of the colors of the connecting lines, those correlations are more pronounced than those in the text. Here's our version of Figure 13.10.b.

```

# new data for `fitted()`
nd <-
tibble(logpop = seq(from = 6, to = 14, length.out = 30),
      lat     = median(d$lat),
      lon2    = median(d$lon2))

# `fitted()`
f <-
fitted(b13.7, newdata = nd) %>%
as_tibble() %>%
bind_cols(nd)

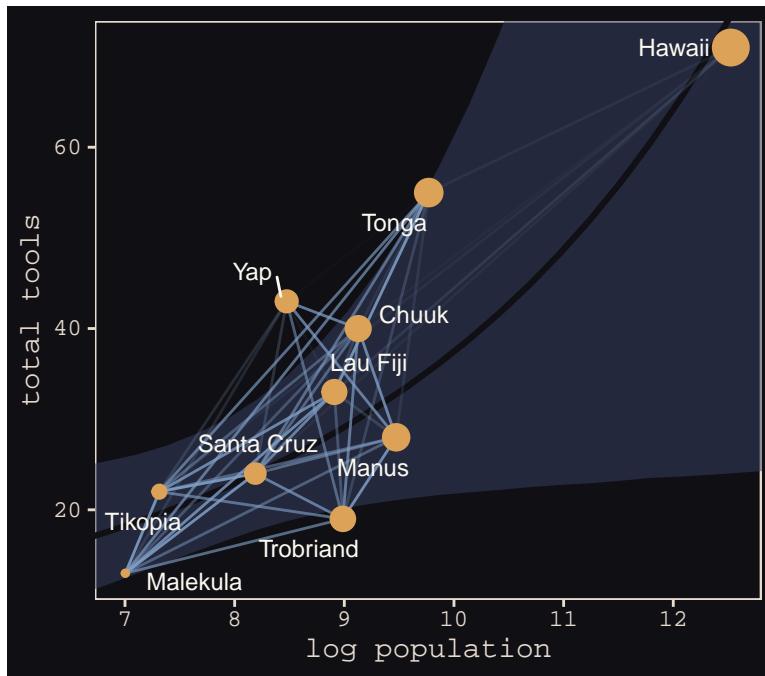
# plot
tidy_rho %>%
ggplot(aes(x = logpop)) +
geom_smooth(data = f,
            aes(y = Estimate, ymin = Q2.5, ymax = Q97.5),
            stat = "identity",
            fill = "#394165", color = "#100F14", alpha = .5, size = 1.1) +
geom_line(aes(y = total_tools, group = group, alpha = correlation^2),
          color = "#80A0C7") +
geom_point(data = d,

```

```

aes(y = total_tools, size = logpop), color = "#DCA258") +
geom_text_repel(data = d,
                 aes(y = total_tools, label = culture),
                 seed = 0, point.padding = .3, size = 3, color = "#FCF9F0") +
scale_alpha_continuous(range = c(0, 1)) +
labs(x = "log population",
     y = "total tools") +
coord_cartesian(xlim = range(d$logpop),
                 ylim = range(d$total_tools)) +
theme(legend.position = "none") +
theme_pearl_earring

```



Same deal. Our higher correlations make for a more intensely-webbed plot. To learn more on Bürkner's thoughts on this model in brms, check out the [thread on this issue](#).

13.5 Summary Bonus: Another Berkley-admissions-data-like example.

McElreath uploaded recordings of him teaching out of his text for a graduate course during the 2017/2018 fall semester. In the beginning of [lecture 13 from week 7](#), he discussed a paper from [van der Lee and Ellemers \(2015\)](#) published in [PNAS](#). Their paper suggested male researchers were more likely than female researchers to get research funding in the Netherlands. In their initial analysis (p. 12350) they provided a simple χ^2 test to test the null hypothesis there was no difference in success for male versus female researchers, for which they reported $\chi^2_{df=1} = 4.01, p = .045$. Happily, van der Lee and Ellemers provided their data values in their supplemental material (i.e., [Table S1](#)), which McElreath also displayed in his video.

Their data follows the same structure as the Berkley admissions data. In his lecture, McElreath suggested their χ^2 test is an example of Simpson's paradox, just as with the Berkley data. He isn't the first person to raise this criticism (see [Volker and SteenBeek's critique](#), which McElreath also pointed to in the lecture).

Here are the data:

```

funding <-
tibble(
  discipline  = rep(c("Chemical sciences", "Physical sciences", "Physics", "Humanities",
                      "Technical sciences", "Interdisciplinary", "Earth/life sciences",
                      "Social sciences", "Medical sciences"),
  each = 2),

```

```

gender      = rep(c("m", "f"), times = 9),
applications = c(83, 39, 135, 39, 67, 9, 230, 166, 189, 62, 105, 78, 156, 126, 425, 409, 245, 260) %>% as.
awards      = c(22, 10, 26, 9, 18, 2, 33, 32, 30, 13, 12, 17, 38, 18, 65, 47, 46, 29) %>% as.integer(),
rejects     = c(61, 29, 109, 30, 49, 7, 197, 134, 159, 49, 93, 61, 118, 108, 360, 362, 199, 231) %>% as.integer()
male        = ifelse(gender == "f", 0, 1) %>% as.integer()
)

funding
## # A tibble: 18 x 6
##   discipline gender applications awards rejects male
##   <chr>      <chr>          <int>    <int>    <int> <int>
## 1 Chemical sciences m              83       22      61     1
## 2 Chemical sciences f              39       10      29     0
## 3 Physical sciences m             135       26     109     1
## 4 Physical sciences f              39       9      30     0
## 5 Physics m                 67       18      49     1
## 6 Physics f                 9        2       7     0
## 7 Humanities m                230       33     197     1
## 8 Humanities f                166       32     134     0
## 9 Technical sciences m             189       30     159     1
## 10 Technical sciences f               62       13      49     0
## 11 Interdisciplinary m              105       12      93     1
## 12 Interdisciplinary f               78       17      61     0
## 13 Earth/life sciences m             156       38     118     1
## 14 Earth/life sciences f               126       18     108     0
## 15 Social sciences m                425       65     360     1
## 16 Social sciences f                409       47     362     0
## 17 Medical sciences m                245       46     199     1
## 18 Medical sciences f                260       29     231     0

```

Let's fit a few models.

First, we'll fit an analogue to the initial van der Lee and Ellemers χ^2 test. Since we're Bayesian modelers, we'll use a simple logistic regression, using `male` (dummy coded 0 = female, 1 = male) to predict admission (i.e., `awards`).

```

b13.bonus_0 <-
  brm(data = funding, family = binomial,
       awards | trials(applications) ~ 1 + male,
       # note our continued use of weakly-regularizing priors
       prior = c(prior(normal(0, 4), class = Intercept),
                  prior(normal(0, 4), class = b)),
       iter = 5000, warmup = 1000, chains = 4, cores = 4,
       seed = 13)

```

If you inspect them, the chains look great. Here are the posterior summaries:

```

tidy(b13.bonus_0) %>%
  filter(term != "lp__") %>%
  mutate_if(is.numeric, round, digits = 2)

##           term estimate std.error lower upper
## 1 b_Intercept -1.75     0.08 -1.88 -1.62
## 2 b_male       0.21     0.10  0.04  0.38

```

Yep, the 95% intervals for `male` dummy exclude zero. If you wanted a one-sided Bayesian p -value, you might do something like:

```
posterior_samples(b13.bonus_0) %>%
  summarise(one_sided_Bayesian_p_value = mean(b_male <= 0))

##   one_sided_Bayesian_p_value
## 1                 0.0181875
```

Pretty small. But recall how Simpson's paradox helped us understand the Berkley data. Different departments in Berkley had different acceptance rates AND different ratios of male and female applicants. Similarly, different academic disciplines in the Netherlands might have different award rates for funding AND different ratios of male and female applications.

Just like in section 13.2, let's fit two more models. The first model will allow intercepts to vary by discipline. The second model will allow intercepts and the `male` dummy slopes to vary by discipline.

```
b13.bonus_1 <-
  brm(data = funding, family = binomial,
       awards | trials(applications) ~ 1 + male + (1 | discipline),
       prior = c(prior(normal(0, 4), class = Intercept),
                 prior(normal(0, 4), class = b),
                 prior(cauchy(0, 1), class = sd)),
       iter = 5000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = .99),
       seed = 13)

b13.bonus_2 <-
  brm(data = funding, family = binomial,
       awards | trials(applications) ~ 1 + male + (1 + male | discipline),
       prior = c(prior(normal(0, 4), class = Intercept),
                 prior(normal(0, 4), class = b),
                 prior(cauchy(0, 1), class = sd),
                 prior(lkj(4), class = cor)),
       iter = 5000, warmup = 1000, chains = 4, cores = 4,
       control = list(adapt_delta = .99),
       seed = 13)
```

We'll compare the models with information criteria.

```
b13.bonus_0 <- add_criterion(b13.bonus_0, "waic")
b13.bonus_1 <- add_criterion(b13.bonus_1, "waic")
b13.bonus_2 <- add_criterion(b13.bonus_2, "waic")

loo_compare(b13.bonus_0, b13.bonus_1, b13.bonus_2,
            criterion = "waic") %>%
  print(simplify = F)

##           elpd_diff se_diff elpd_waic se_elpd_waic p_waic se_p_waic waic    se_waic
## b13.bonus_2    0.0      0.0    -58.3      2.8     8.8     1.0    116.6    5.6
## b13.bonus_1   -4.6      1.4    -62.9      3.7    10.1     1.5    125.9    7.3
## b13.bonus_0   -6.5      2.7    -64.8      4.4     4.6     1.3    129.5    8.8
```

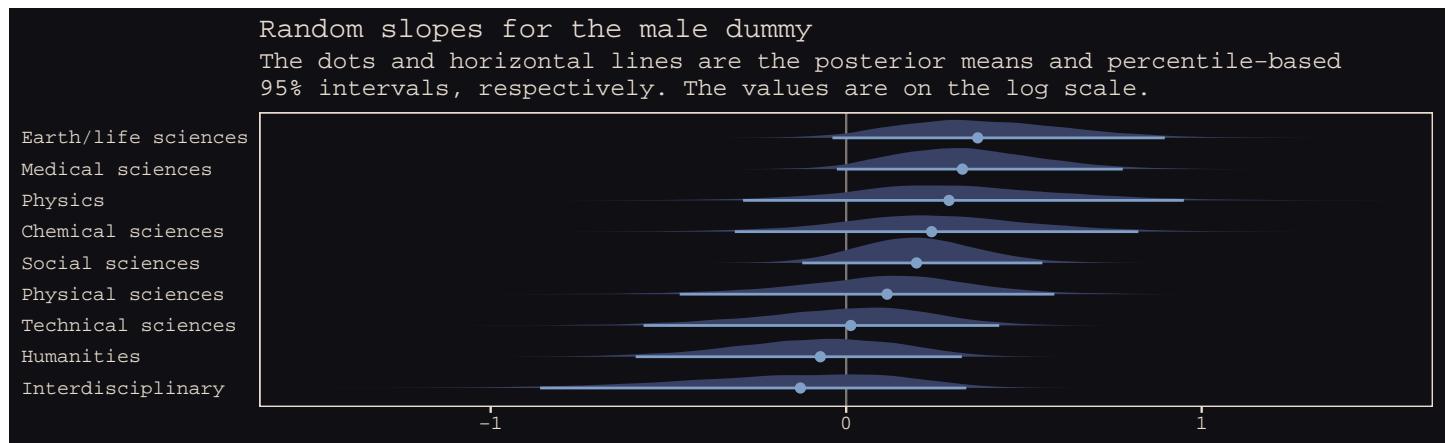
The WAIC suggests the varying intercepts/varying slopes model made the best sense of the data. Here we'll practice our `tidybayes::spread_draws()` skills from the end of the last chapter to see what the random intercepts look like in a coefficient plot.

```
b13.bonus_2 %>%
  spread_draws(b_male, r_discipline[discipline,term]) %>%
  filter(term == "male") %>%
  ungroup() %>%
  mutate(effect      = b_male + r_discipline,
```

```

discipline = str_replace(discipline, "[.]", " ") %>%
  ggplot(aes(x = effect, y = reorder(discipline, effect))) +
  geom_vline(xintercept = 0, color = "#E8DCCF", alpha = 1/2) +
  geom_halfeyeh(.width = .95, size = .9, relative_scale = .9,
                 color = "#80A0C7", fill = "#394165") +
  labs(title = "Random slopes for the male dummy",
       subtitle = "The dots and horizontal lines are the posterior means and percentile-based\n95% intervals, respectively. The values are on the log scale.",
       x = NULL, y = NULL) +
  coord_cartesian(xlim = c(-1.5, 1.5)) +
  theme_pearl_earring +
  theme(axis.ticks.y = element_blank(),
        axis.text.y = element_text(hjust = 0))

```



Note how the 95% intervals for all the random `male` slopes contain zero within their bounds. Here are the fixed effects:

```

tidy(b13.bonus_2) %>%
  filter(str_detect(term , "b_")) %>%
  mutate_if(is.numeric, round, digits = 2)

##           term estimate std.error lower upper
## 1 b_Intercept   -1.62      0.14 -1.84 -1.38
## 2     b_male      0.15      0.17 -0.14  0.42

```

And if you wanted a one-sided Bayesian p -value for the `male` dummy for the full model:

```

posterior_samples(b13.bonus_2) %>%
  summarise(one_sided_Bayesian_p_value = mean(b_male <= 0))

```

```

##   one_sided_Bayesian_p_value
## 1                      0.1708125

```

So, the estimate of the gender bias is small and consistent with the null hypothesis. Which is good! We want gender equality for things like funding success.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods  base
##
## other attached packages:
## [1] brms_2.8.8          bayesplot_1.6.0    ggbeeswarm_0.6.0   ggrepel_0.8.0
## [5] tidybayes_1.0.4     broom_0.5.1       Rcpp_1.0.1         rstan_2.18.2
## [9] StanHeaders_2.18.0-1 dutchmasters_0.1.0 forcats_0.3.0     stringr_1.4.0
## [13] dplyr_0.8.0.1      purrr_0.2.5       readr_1.1.1        tidyr_0.8.1
## [17] tibble_2.1.1        ggplot2_3.1.1     tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2      ggridges_0.5.0      rsconnect_0.8.8
## [4] rprojroot_1.3-2       ggstance_0.3        markdown_0.8
## [7] base64enc_0.1-3       rstudioapi_0.7       svUnit_0.7-12
## [10] DT_0.4                fansi_0.4.0        mvtnorm_1.0-10
## [13] lubridate_1.7.4      xml2_1.2.0         bridgesampling_0.6-0
## [16] knitr_1.20             shinythemes_1.1.1   jsonlite_1.5
## [19] shiny_1.1.0            compiler_3.5.1     httr_1.3.1
## [22] backports_1.1.4       assertthat_0.2.0   Matrix_1.2-14
## [25] lazyeval_0.2.2        cli_1.0.1          later_0.7.3
## [28] htmltools_0.3.6       prettyunits_1.0.2  tools_3.5.1
## [31] igraph_1.2.1           coda_0.19-2        gtable_0.3.0
## [34] glue_1.3.1.9000       reshape2_1.4.3     cellranger_1.1.0
## [37] nlme_3.1-137          crosstalk_1.0.0   xfun_0.3
## [40] ps_1.2.1              rvest_0.3.2        mime_0.5
## [43] miniUI_0.1.1.1        gtools_3.8.1       nleqslv_3.3.2
## [46] MASS_7.3-50            zoo_1.8-2          scales_1.0.0
## [49] colourpicker_1.0       hms_0.4.2          promises_1.0.1
## [52] Broddingnag_1.2-6     inline_0.3.15     shinystan_2.5.0
## [55] yaml_2.1.19            gridExtra_2.3     loo_2.1.0
## [58] stringi_1.4.3          dygraphs_1.1.1.5  pkgbuild_1.0.2
## [61] rlang_0.3.4             pkgconfig_2.0.2   matrixStats_0.54.0
## [64] HDInterval_0.2.0       evaluate_0.10.1   lattice_0.20-35
## [67] rstantools_1.5.1       htmlwidgets_1.2    labeling_0.3
## [70] processx_3.2.1         tidyselect_0.2.5  plyr_1.8.4
## [73] magrittr_1.5            bookdown_0.9      R6_2.3.0
## [76] generics_0.0.2          pillar_1.3.1     haven_1.1.2
## [79] withr_2.1.2             xts_0.10-2        abind_1.4-5
## [82] modelr_0.1.2            crayon_1.3.4     arrayhelpers_1.0-20160527
## [85] utf8_1.1.4              rmarkdown_1.10   grid_3.5.1
## [88] readxl_1.1.0             callr_3.1.0      threejs_0.3.1
## [91] digest_0.6.18            xtable_1.8-2     httpuv_1.4.4.2
## [94] stats4_3.5.1             munsell_0.5.0    beeswarm_0.2.3
## [97] vipor_0.4.5             shinyjs_1.0
```

Chapter 14

Missing Data and Other Opportunities

For the opening example, we're playing with the conditional probability

$$\Pr(\text{burnt down} \mid \text{burnt up}) = \frac{\Pr(\text{burnt up, burnt down})}{\Pr(\text{burnt up})}$$

It works out that

$$\Pr(\text{burnt down} \mid \text{burnt up}) = \frac{1/3}{1/2} = \frac{2}{3}$$

We might express the math in the middle of page 423 in tibble form like this.

```
library(tidyverse)

p_pancake <- 1/3

(
  d <-
    tibble(pancake      = c("BB", "BU", "UU"),
           p_burnt       = c(1, .5, 0)) %>%
    mutate(p_burnt_up = p_burnt * p_pancake)
)

## # A tibble: 3 x 3
##   pancake p_burnt p_burnt_up
##   <chr>     <dbl>     <dbl>
## 1 BB         1        0.333
## 2 BU         0.5      0.167
## 3 UU         0        0

d %>%
  summarise(`p (burnt_down | burnt_up)` = p_pancake / sum(p_burnt_up))

## # A tibble: 1 x 1
##   `p (burnt_down | burnt_up)`
##   <dbl>
## 1 0.667
```

I understood McElreath's simulation better after breaking it apart. The first part of `sim_pancake()` takes one random draw from the integers 1, 2, and 3. It just so happens that if we set `set.seed(1)`, the code returns a 1.

```
set.seed(1)
sample(x = 1:3, size = 1)

## [1] 1
```

So here's what it looks like if we use seeds 2:11.

```
take_sample <- function(seed){
  set.seed(seed)
  sample(x = 1:3, size = 1)
}

tibble(seed = 2:11) %>%
  mutate(value_returned = map_dbl(seed, take_sample))
```

```
## # A tibble: 10 x 2
##   seed value_returned
##   <int>      <dbl>
## 1     2          1
## 2     3          1
## 3     4          2
## 4     5          1
## 5     6          2
## 6     7          3
## 7     8          2
## 8     9          1
## 9    10          2
## 10   11          1
```

Each of those `value_returned` values stands for one of the three pancakes: 1 = BB, 2 = BU, 3 = UU. In the next line, McElreath made slick use of a matrix to specify that. Here's what the matrix looks like:

```
matrix(c(1, 1, 1, 0, 0, 0), nrow = 2, ncol = 3)

##      [,1] [,2] [,3]
## [1,]    1    1    0
## [2,]    1    0    0
```

See how the three columns are identified as `[,1]`, `[,2]`, and `[,3]`? If, say, we wanted to subset the values in the second column, we'd execute

```
matrix(c(1, 1, 1, 0, 0, 0), nrow = 2, ncol = 3)[, 2]

## [1] 1 0
```

which returns a numeric vector.

```
matrix(c(1, 1, 1, 0, 0, 0), nrow = 2, ncol = 3)[, 2] %>% str()

## num [1:2] 1 0
```

And that 1 0 corresponds to the pancake with one burnt (i.e., 1) and one unburnt (i.e., 0) side. So when McElreath then executed `sample(sides)`, he randomly sampled from one of those two values. In the case of `pancake == 2`, he randomly sampled one the pancake with one burnt and one unburnt side. Had he sampled from `pancake == 1`, he would have sampled from the pancake with both sides burnt.

Going forward, let's amend McElreath's `sim_pancake()` function a bit. First, we'll add a `seed` argument, which will allow us to make the output reproducible. We'll be inserting `seed` into `set.seed()` in the two places preceding the `sample()` function. The second major change is that we're going to convert the output of the `sim_pancake()` function to a tibble and adding a `side` column, which will contain the values `c("up", "down")`. Just for pedagogical purposes, we'll also add `pancake_n` and `pancake_chr` columns to help index which pancake the draws came from.

```
# simulate a `pancake` and return randomly ordered `sides`
sim_pancake <- function(seed) {
  set.seed(seed)
  pancake <- sample(x = 1:3, size = 1)
  sides   <- matrix(c(1, 1, 1, 0, 0, 0), nrow = 2, ncol = 3)[, pancake]

  set.seed(seed)
  sample(sides) %>%
    as_tibble() %>%
    mutate(side      = c("up", "down"),
          pancake_n = pancake,
          pancake_chr = ifelse(pancake == 1, "BB",
                               ifelse(pancake == 2, "BU", "UU")))
}
```

Let's take this baby for a whirl.

```
# how many simulations would you like?
n_sim <- 1e4

(
  d <-
  tibble(seed = 1:n_sim) %>%
  mutate(r = map(seed, sim_pancake)) %>%
  unnest()
)

## # A tibble: 20,000 x 5
##   seed value side  pancake_n pancake_chr
##   <int> <dbl> <chr>    <int> <chr>
## 1     1     1 up        1 BB
## 2     1     1 down     1 BB
## 3     2     1 up        1 BB
## 4     2     1 down     1 BB
## 5     3     1 up        1 BB
## 6     3     1 down     1 BB
## 7     4     0 up        2 BU
## 8     4     1 down     2 BU
## 9     5     1 up        1 BB
## 10    5     1 down     1 BB
## # ... with 19,990 more rows
```

And now we'll `spread()` and `summarise()` to get the value we've been working for.

```
d %>%
  spread(key = side, value = value) %>%
  summarise(`p (burnt_down | burnt_up)` = sum(up == 1 & down == 1) / (sum(up == 1)))

## # A tibble: 1 x 1
##   `p (burnt_down | burnt_up)`
##   <dbl>
## 1 0.661
```

The results are within rounding error of the ideal 2/3.

Probability theory is not difficult mathematically. It's just counting. But it is hard to interpret and apply. Doing so often seems to require some cleverness, and authors have an incentive to solve problems in clever ways, just to show off. But we don't need that cleverness, if we ruthlessly apply conditional probability...

In this chapter, [we'll] meet two commonplace applications of this assume-and-deduce strategy. The first is the incorporation of measurement error into our models. The second is the estimation of missing data through Bayesian imputation...

In neither application do [we] have to intuit the consequences of measurement errors nor the implications of missing values in order to design the models. All [we] have to do is state [the] information about the error or about the variables with missing values. Logic does the rest. (p. 424)

14.1 Measurement error

First, let's grab our `WaffleDivorce` data.

```
library(rethinking)
data(WaffleDivorce)
d <- WaffleDivorce
rm(WaffleDivorce)
```

Switch out `rethinking` for `brms`.

```
detach(package:rethinking, unload = T)
library(brms)
```

```
## Warning: package 'Rcpp' was built under R version 3.5.2
```

The `brms` package currently supports `theme_black()`, which changes the default `ggplot2` theme to a black background with white lines, text, and so forth. You can find the origins of the code, [here](#).

Though I like the idea of `brms` including `theme_black()`, I'm not a fan of some of the default settings (e.g., it includes gridlines). Happily, data scientist [Tyler Rinker](#) has some nice alternative `theme_black()` code you can find [here](#). The version of `theme_black()` used for this chapter is based on his version, with a few amendments of my own.

```
theme_black <-
  function(base_size=12, base_family="") {
    theme_grey(base_size=base_size, base_family=base_family) %+replace%
    theme(
      # specify axis options
      axis.line=element_blank(),
      # all text colors used to be "grey55"
      axis.text.x=element_text(size=base_size*0.8, color="grey85",
                               lineheight=0.9, vjust=1),
      axis.text.y=element_text(size=base_size*0.8, color="grey85",
                               lineheight=0.9,hjust=1),
      axis.ticks=element_line(color="grey55", size = 0.2),
      axis.title.x=element_text(size=base_size, color="grey85", vjust=1,
                                margin=ggplot2::margin(.5, 0, 0, 0, "lines")),
      axis.title.y=element_text(size=base_size, color="grey85", angle=90,
                                margin=ggplot2::margin(.5, 0, 0, 0, "lines"), vjust=0.5),
      axis.ticks.length=grid::unit(0.3, "lines"),
      # specify legend options
      legend.background=element_rect(color=NA, fill="black"),
      legend.key=element_rect(color="grey55", fill="black"),
      legend.key.size=grid::unit(1.2, "lines"),
```

```

legend.key.height=NULL,
legend.key.width=NULL,
legend.text=element_text(size=base_size*0.8, color="grey85"),
legend.title=element_text(size=base_size*0.8, face="bold", hjust=0,
    color="grey85"),
# legend.position="right",
legend.position = "none",
legend.text.align=NULL,
legend.title.align=NULL,
legend.direction="vertical",
legend.box=NULL,
# specify panel options
panel.background=element_rect(fill="black", color = NA),
panel.border=element_rect(fill=NA, color="grey55"),
panel.grid.major=element_blank(),
panel.grid.minor=element_blank(),
panel.spacing=grid::unit(0.25,"lines"),
# specify facetting options
strip.background=element_rect(fill = "black", color="grey10"), # fill="grey30"
strip.text.x=element_text(size=base_size*0.8, color="grey85"),
strip.text.y=element_text(size=base_size*0.8, color="grey85",
    angle=-90),
# specify plot options
plot.background=element_rect(color="black", fill="black"),
plot.title=element_text(size=base_size*1.2, color="grey85", hjust = 0), # added hjust = 0
plot.subtitle=element_text(size=base_size*.9, color="grey85", hjust = 0), # added line
# plot.margin=grid::unit(c(1, 1, 0.5, 0.5), "lines")
plot.margin=grid::unit(c(0.5, 0.5, 0.5, 0.5), "lines")
)
}

```

One way to use our `theme_black()` is to make it part of the code for an individual plot, such as `ggplot() + geom_point() + theme_black()`. Another way is to make `theme_black()` the default setting with `ggplot2::theme_set()`. That's the method we'll use.

```

theme_set(theme_black())

# to reset the default ggplot2 theme to its default parameters,
# execute `theme_set(theme_default())` 

```

In the [brms reference manual](#), Bürkner recommended complimenting `theme_black()` with color scheme “C” from the `viridis` package, which provides a variety of colorblind-safe color palettes.

```

# install.packages("viridis")
library(viridis)

```

The `viridis_pal()` function gives a list of colors within a given palette. The colors in each palette fall on a spectrum. Within `viridis_pal()`, the `option` argument allows one to select a given spectrum, “C”, in our case. The final parentheses, `()`, allows one to determine how many discrete colors one would like to break the spectrum up by. We'll choose 7.

```

viridis_pal(option = "C")(7)

## [1] "#D0887FF" "#5D01A6FF" "#9C179EFF" "#CC4678FF" "#ED7953FF" "#FDB32FFF" "#F0F921FF"

```

With a little data wrangling, we can put the colors of our palette in a tibble and display them in a plot.

```
tibble(number      = 1:7,
       color_number = str_c(1:7, ". ", viridis_pal(option = "C")(7))) %>%
  ggplot(aes(x = factor(0), y = reorder(color_number, number))) +
  geom_tile(aes(fill = factor(number))) +
  geom_text(aes(color = factor(number), label = color_number)) +
  scale_color_manual(values = c(rep("black", times = 4),
                                 rep("white", times = 3))) +
  scale_fill_viridis(option = "C", discrete = T, direction = -1) +
  scale_x_discrete(NULL, breaks = NULL) +
  scale_y_discrete(NULL, breaks = NULL) +
  ggtile("Behold: viridis C!")
```

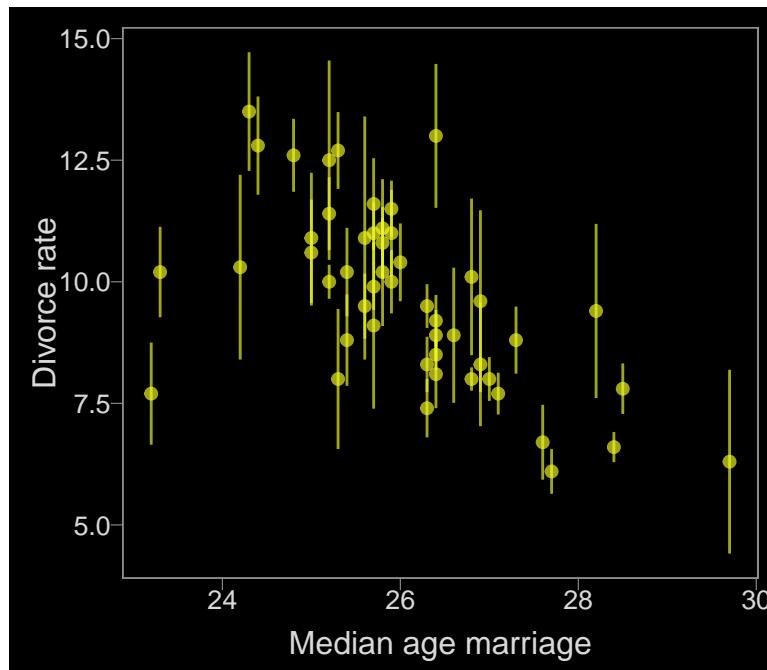
Behold: viridis C!



Now, let's make use of our custom theme and reproduce/reimagine Figure 14.1.a.

```
color <- viridis_pal(option = "C")(7) [7]

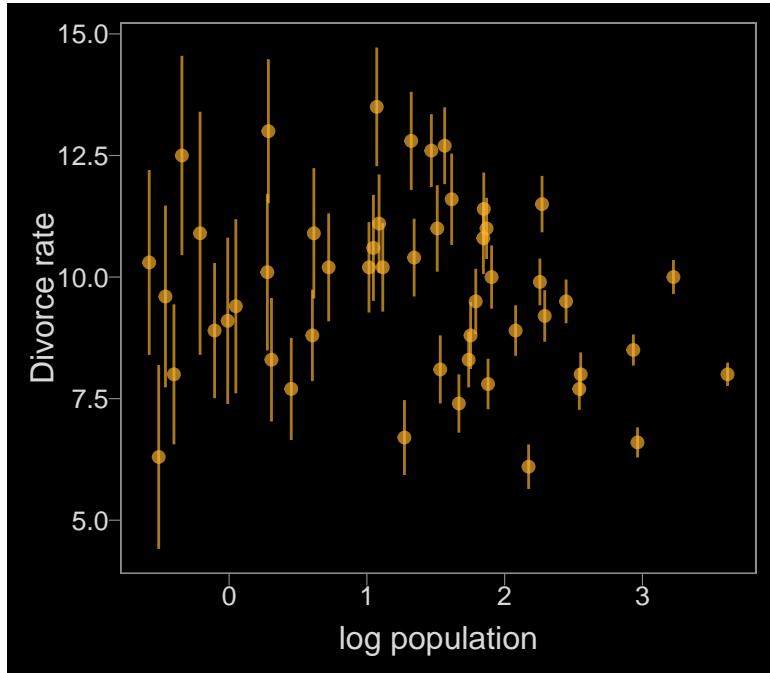
d %>%
  ggplot(aes(x = MedianAgeMarriage,
             y = Divorce,
             ymin = Divorce - Divorce.SE,
             ymax = Divorce + Divorce.SE)) +
  geom_pointrange(shape = 20, alpha = 2/3, color = color) +
  labs(x = "Median age marriage" ,
       y = "Divorce rate")
```



Notice how `viridis_pal(option = "C")(7)[7]` called the seventh color in the color scheme, "#F0F921FF". For Figure 14.1.b, we'll select the sixth color in the palette by coding `viridis_pal(option = "C")(7)[6]`.

```
color <- viridis_pal(option = "C")(7)[6]

d %>%
  ggplot(aes(x = log(Population),
             y = Divorce,
             ymin = Divorce - Divorce.SE,
             ymax = Divorce + Divorce.SE)) +
  geom_pointrange(shape = 20, alpha = 2/3, color = color) +
  labs(x = "log population",
       y = "Divorce rate")
```



Just like in the text, our plot shows states with larger populations tend to have smaller measurement error.

14.1.1 Error on the outcome.

To get a better sense of what we're about to do, imagine for a moment that each state's divorce rate is normally distributed with a mean of `Divorce` and standard deviation `Divorce.SE`. Those distributions would be:

```
d %>%
  mutate(Divorce_distribution = str_c("Divorce ~ Normal(", Divorce, ", ", Divorce.SE, ")")) %>%
  select(Loc, Divorce_distribution) %>%
  head()
```

```
##   Loc      Divorce_distribution
## 1 AL Divorce ~ Normal(12.7, 0.79)
## 2 AK Divorce ~ Normal(12.5, 2.05)
## 3 AZ Divorce ~ Normal(10.8, 0.74)
## 4 AR Divorce ~ Normal(13.5, 1.22)
## 5 CA   Divorce ~ Normal(8, 0.24)
## 6 CO Divorce ~ Normal(11.6, 0.94)
```

As in the text,

in [the following] example we'll use a Gaussian distribution with mean equal to the observed value and standard deviation equal to the measurement's standard error. This is the logical choice, because if all we know about the error is its standard deviation, then the maximum entropy distribution for it will be Gaussian...

Here's how to define the distribution for each divorce rate. For each observed value D_{OBS_i} , there will be one parameter, D_{EST_i} , defined by:

$$D_{\text{OBS}_i} \sim \text{Normal}(D_{\text{EST}_i}, D_{\text{SE}_i})$$

All this does is define the measurement D_{OBS_i} as having the specified Gaussian distribution centered on the unknown parameter D_{EST_i} . So the above defines a probability for each State i 's observed divorce rate, given a known measurement error. (pp. 426–427)

Now we're ready to fit some models. In brms, there are at least two ways to accommodate measurement error in the criterion. The first way uses the `se()` syntax, following the form `<response> | se(<se_response>, sigma = TRUE)`. With this syntax, `se` stands for standard error, the loose frequentist analogue to the Bayesian posterior SD . Unless you're [fitting a meta-analysis](#) on summary information, which we'll be doing at the end of this chapter, make sure to specify `sigma = TRUE`. Without that you'll have no estimate for $\sigma!$ For more information on the `se()` method, go to the [brms reference manual](#) and find the *Additional response information* subsection of the `brmsformula` section.

The second way uses the `mi()` syntax, following the form `<response> | mi(<se_response>)`. This follows a missing data logic, resulting in Bayesian missing data imputation for the criterion values. The `mi()` syntax is based on the newer missing data capabilities for brms. We will cover that in more detail in the second half of this chapter.

We'll start off using both methods. Our first model, `b14.1_se`, will follow the `se()` syntax; the second model, `b14.1_mi`, will follow the `mi()` syntax.

```
# put the data into a `list()`
dlist <- list(
  div_obs = d$Divorce,
  div_sd  = d$Divorce.SE,
  R       = d$Marriage,
  A       = d$MedianAgeMarriage)

# here we specify the initial (i.e., starting) values
inits     <- list(Y1 = dlist$div_obs)
inits_list <- list(inits, inits)

# fit the models
b14.1_se <-
  brm(data = dlist, family = gaussian,
    div_obs | se(div_sd, sigma = TRUE) ~ 0 + intercept + R + A,
    prior = c(prior(normal(0, 10), class = b),
              prior(cauchy(0, 2.5), class = sigma)),
    iter = 5000, warmup = 1000, cores = 2, chains = 2,
    seed = 14,
    control = list(adapt_delta = 0.99,
                   max_treedepth = 12),
    inits = inits_list)

b14.1_mi <-
  brm(data = dlist, family = gaussian,
    div_obs | mi(div_sd) ~ 0 + intercept + R + A,
    prior = c(prior(normal(0, 10), class = b),
              prior(cauchy(0, 2.5), class = sigma)),
    iter = 5000, warmup = 1000, cores = 2, chains = 2,
    seed = 14,
    control = list(adapt_delta = 0.99,
                   max_treedepth = 12),
    save_mevars = TRUE, # note this line for the `mi()` model
    inits = inits_list)
```

Before we dive into the model summaries, notice how the starting values (i.e., `inits`) differ by model. Even though we coded `inits = inits_list` for both models, the differ by `fit@inits`.

```
b14.1_se$fit@inits
```

```
## [[1]]
## [[1]]$b
## [1] 0.6133048 -1.9171497 1.7551789
##
## [[1]]$sigma
## [1] 0.4668127
##
## 
## [[2]]
## [[2]]$b
## [1] 0.9114156 1.2512265 -0.4276127
##
## [[2]]$sigma
## [1] 1.906943
```

```
b14.1_mi$fit@inits
```

```
## [[1]]
## [[1]]$Y1
## [1] 12.7 12.5 10.8 13.5 8.0 11.6 6.7 8.9 6.3 8.5 11.5 8.3 7.7 8.0 11.0 10.2 10.6 12.6 11.0
## [20] 13.0 8.8 7.8 9.2 7.4 11.1 9.5 9.1 8.8 10.1 6.1 10.2 6.6 9.9 8.0 9.5 12.8 10.4 7.7
## [39] 9.4 8.1 10.9 11.4 10.0 10.2 9.6 8.9 10.0 10.9 8.3 10.3
##
## [[1]]$b
## [1] -0.5034648 1.1693530 -1.0539336
##
## [[1]]$sigma
## [1] 1.281562
##
## 
## [[2]]
## [[2]]$Y1
## [1] 12.7 12.5 10.8 13.5 8.0 11.6 6.7 8.9 6.3 8.5 11.5 8.3 7.7 8.0 11.0 10.2 10.6 12.6 11.0
## [20] 13.0 8.8 7.8 9.2 7.4 11.1 9.5 9.1 8.8 10.1 6.1 10.2 6.6 9.9 8.0 9.5 12.8 10.4 7.7
## [39] 9.4 8.1 10.9 11.4 10.0 10.2 9.6 8.9 10.0 10.9 8.3 10.3
##
## [[2]]$b
## [1] -0.1543955 1.1642108 -0.4231833
##
## [[2]]$sigma
## [1] 4.802142
```

As we explore further, it should become apparent why. Here are the primary model summaries.

```
print(b14.1_se)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: div_obs | se(div_sd, sigma = TRUE) ~ 0 + intercept + R + A
##   Data: dlist (Number of observations: 50)
## Samples: 2 chains, each with iter = 5000; warmup = 1000; thin = 1;
##           total post-warmup samples = 8000
##
```

```

## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## intercept    21.21      6.62     7.81   33.76      1889 1.00
## R            0.13      0.08    -0.02    0.28      2230 1.00
## A           -0.55      0.21    -0.94   -0.11      2040 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.13      0.21     0.77    1.57      3182 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

```
print(b14.1_mi)
```

```

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: div_obs | mi(div_sd) ~ 0 + intercept + R + A
##   Data: dlist (Number of observations: 50)
## Samples: 2 chains, each with iter = 5000; warmup = 1000; thin = 1;
##           total post-warmup samples = 8000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## intercept    21.37      6.52     8.50   33.82      3831 1.00
## R            0.13      0.08    -0.02    0.27      4493 1.00
## A           -0.55      0.21    -0.95   -0.13      3924 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.13      0.21     0.76    1.56      3200 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).

```

Based on the `print()`/`summary()` information, the main parameters for the models are about the same. However, the plot deepens when we summarize the models with the `broom::tidy()` method.

```

library(broom)

tidy(b14.1_se) %>%
  mutate_if(is.numeric, round, digits = 2)

```

```

##          term estimate std.error   lower   upper
## 1 b_intercept    21.21      6.62  10.19  31.85
## 2 b_R            0.13      0.08   0.00   0.26
## 3 b_A           -0.55      0.21  -0.89  -0.19
## 4 sigma          1.13      0.21   0.82   1.49
## 5 lp__         -105.41     1.42 -108.18 -103.71

```

```

tidy(b14.1_mi) %>%
  mutate_if(is.numeric, round, digits = 2)

```

```

##          term estimate std.error   lower   upper
## 1 b_intercept    21.37      6.52  10.68  31.79
## 2 b_R            0.13      0.08   0.00   0.25

```

```

## 3      b_A    -0.55    0.21   -0.89   -0.20
## 4      sigma   1.13    0.21    0.80    1.48
## 5      Y1[1]   11.79    0.68   10.70   12.92
## 6      Y1[2]   11.20    1.04   9.53    12.92
## 7      Y1[3]   10.47    0.63   9.44    11.51
## 8      Y1[4]   12.31    0.85   10.94   13.74
## 9      Y1[5]    8.05    0.24   7.66    8.44
## 10     Y1[6]   11.02    0.74   9.82    12.25
## 11     Y1[7]    7.23    0.64   6.17    8.27
## 12     Y1[8]    9.35    0.91   7.84    10.82
## 13     Y1[9]    7.00    1.09   5.22    8.76
## 14     Y1[10]   8.54    0.30   8.04    9.04
## 15     Y1[11]   11.15    0.54   10.27   12.04
## 16     Y1[12]   9.10    0.90   7.60    10.57
## 17     Y1[13]   9.69    0.91   8.15    11.14
## 18     Y1[14]   8.11    0.42   7.43    8.79
## 19     Y1[15]   10.68    0.54   9.79    11.57
## 20     Y1[16]   10.17    0.71   9.02    11.32
## 21     Y1[17]   10.50    0.78   9.22    11.80
## 22     Y1[18]   11.95    0.63   10.94   12.98
## 23     Y1[19]   10.50    0.69   9.39    11.63
## 24     Y1[20]   10.18    1.02   8.53    11.90
## 25     Y1[21]   8.76    0.59   7.78    9.74
## 26     Y1[22]   7.77    0.48   6.98    8.54
## 27     Y1[23]   9.14    0.47   8.36    9.92
## 28     Y1[24]   7.74    0.54   6.84    8.62
## 29     Y1[25]   10.43    0.76   9.20    11.70
## 30     Y1[26]   9.54    0.58   8.58    10.51
## 31     Y1[27]   9.42    0.96   7.84    11.01
## 32     Y1[28]   9.26    0.74   8.05    10.42
## 33     Y1[29]   9.17    0.96   7.62    10.75
## 34     Y1[30]   6.38    0.44   5.64    7.11
## 35     Y1[31]   9.98    0.79   8.66    11.29
## 36     Y1[32]   6.69    0.30   6.21    7.17
## 37     Y1[33]   9.88    0.44   9.17    10.61
## 38     Y1[34]   9.77    0.96   8.15    11.29
## 39     Y1[35]   9.43    0.42   8.75    10.13
## 40     Y1[36]   11.97    0.79   10.70   13.26
## 41     Y1[37]   10.07    0.66   9.00    11.17
## 42     Y1[38]   7.80    0.40   7.14    8.45
## 43     Y1[39]   8.21    1.00   6.62    9.91
## 44     Y1[40]   8.40    0.59   7.41    9.37
## 45     Y1[41]   10.01    1.04   8.30    11.75
## 46     Y1[42]   10.94    0.64   9.88    11.99
## 47     Y1[43]   10.02    0.34   9.46    10.57
## 48     Y1[44]   11.08    0.78   9.79    12.38
## 49     Y1[45]   8.90    0.99   7.31    10.54
## 50     Y1[46]   9.01    0.46   8.24    9.78
## 51     Y1[47]   9.96    0.56   9.04    10.87
## 52     Y1[48]   10.61    0.89   9.16    12.07
## 53     Y1[49]   8.46    0.51   7.63    9.30
## 54     Y1[50]   11.53    1.09   9.67    13.26
## 55     lp__  -152.51    6.58  -163.76  -141.95

```

```
# you can get similar output with `b14.1_mi$fit`
```

Again, from `b_intercept` to `sigma`, the output is about the same. But model `b14.1_mi`, based on the `mi()` syntax, contained posterior summaries for all 50 of the criterion values. The `se()` method gave us similar model result, but no posterior summaries for the 50 criterion values. The `rethinking` package indexed those additional 50 as `div_est[i]`; with the `mi()` method, `brms` indexed them as `Y1[i]`—no big deal. So while both `brms` methods accommodated measurement error,

the `mi()` method appears to be the `brms` analogue to what McElreath did with his model `m14.1` in the text. Thus, it's our `b14.1_mi` model that follows the form

$$\begin{aligned} \text{Divorce}_{\text{estimated},i} &\sim \text{Normal}(\mu_i, \sigma) \\ \mu &= \alpha + \beta_1 A_i + \beta_2 R_i \\ \text{Divorce}_{\text{observed},i} &\sim \text{Normal}(\text{Divorce}_{\text{estimated},i}, \text{Divorce}_{\text{standard error},i}) \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta_1 &\sim \text{Normal}(0, 10) \\ \beta_2 &\sim \text{Normal}(0, 10) \\ \sigma &\sim \text{HalfCauchy}(0, 2.5) \end{aligned}$$

Note. The `normal(0, 10)` prior McElreath used was [quite informative and can lead to discrepancies between the rethinking and brms results](#) if you're not careful. A large issue is the default way `brms` handles intercept priors. From the hyperlink, Bürkner wrote:

The formula for the original intercept is `b_intercept = temp_intercept - dot_product(means_X, b)`, where `means_X` is the vector of means of the predictor variables and `b` is the vector of regression coefficients (fixed effects). That is, when transforming a prior on the intercept to an “equivalent” prior on the temporary intercept, you have to take the means of the predictors and well as the priors on the other coefficients into account.

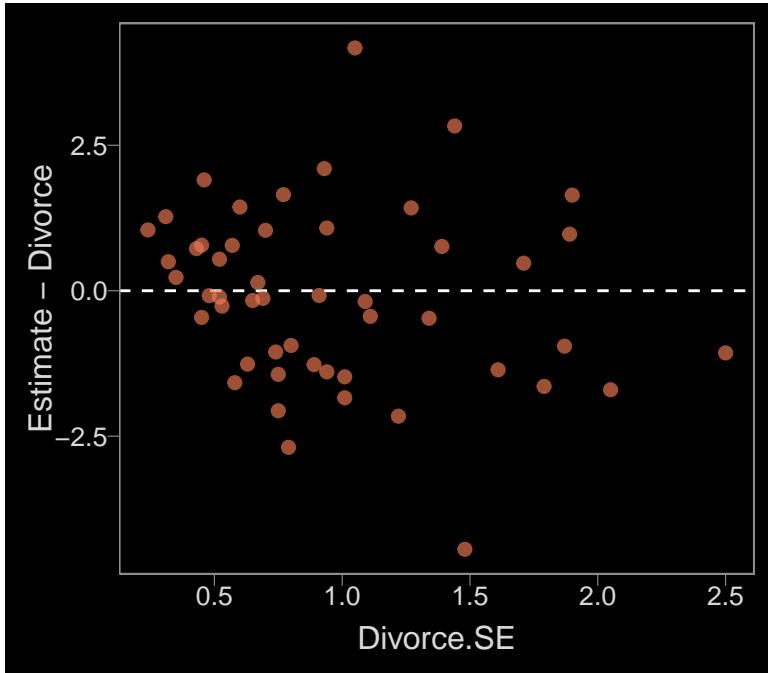
If this seems confusing, you have an alternative. The `0 + intercept` part of the `brm` formula kept the intercept in the metric of the untransformed data, leading to similar results to those from `rethinking`. When your priors are vague, this might not be much of an issue. And since many of the models in *Statistical Rethinking* use only weakly-regularizing priors, this hasn't been much of an issue up to this point. But this model is quite sensitive to the intercept syntax. My general recommendation for applied data analysis is this: **If your predictors aren't mean centered, default to the `0 + intercept` syntax for the formula argument when using `brms::brm()`.** Otherwise, your priors might not be doing what you think they're doing.

Anyway, since our `mi()`-syntax `b14.1_mi` model appears to be the analogue to McElreath's `m14.1`, we'll use that one for our plots. Here's our Figure 14.2.a.

```
data_error <-
  fitted(b14.1_mi) %>%
  as_tibble() %>%
  bind_cols(d)

color <- viridis_pal(option = "C")(7)[5]

data_error %>%
  ggplot(aes(x = Divorce.SE, y = Estimate - Divorce)) +
  geom_hline(yintercept = 0, linetype = 2, color = "white") +
  geom_point(alpha = 2/3, size = 2, color = color)
```



Before we make Figure 14.2.b, we need to fit a model that ignores measurement error.

```
b14.1b <-
  brm(data = dlist, family = gaussian,
    div_obs ~ 0 + intercept + R + A,
    prior = c(prior(normal(0, 50), class = b, coef = intercept),
              prior(normal(0, 10), class = b),
              prior(cauchy(0, 2.5), class = sigma)),
    chains = 2, iter = 5000, warmup = 1000, cores = 2,
    seed = 14,
    control = list(adapt_delta = 0.95))

print(b14.1b)

##  Family: gaussian
##  Links: mu = identity; sigma = identity
##  Formula: div_obs ~ 0 + intercept + R + A
##  Data: dlist (Number of observations: 50)
##  Samples: 2 chains, each with iter = 5000; warmup = 1000; thin = 1;
##          total post-warmup samples = 8000
##
##  Population-Level Effects:
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##  intercept     35.77      7.90    20.42    51.23      2147 1.00
##  R            -0.05      0.08    -0.21     0.12      2439 1.00
##  A            -0.96      0.25    -1.45    -0.47      2243 1.00
##
##  Family Specific Parameters:
##                Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
##  sigma        1.51      0.16    1.23     1.87      2982 1.00
##
##  Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
##  is a crude measure of effective sample size, and Rhat is the potential
##  scale reduction factor on split chains (at convergence, Rhat = 1).
```

With the ignore-measurement-error fit in hand, we're ready for Figure 14.2.b.

```

nd <-
  tibble(R      = mean(d$Marriage),
         A      = seq(from = 22, to = 30.2, length.out = 30),
         div_sd = mean(d$Divorce.SE))

# red line
f_error <-
  fitted(b14.1_mi, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

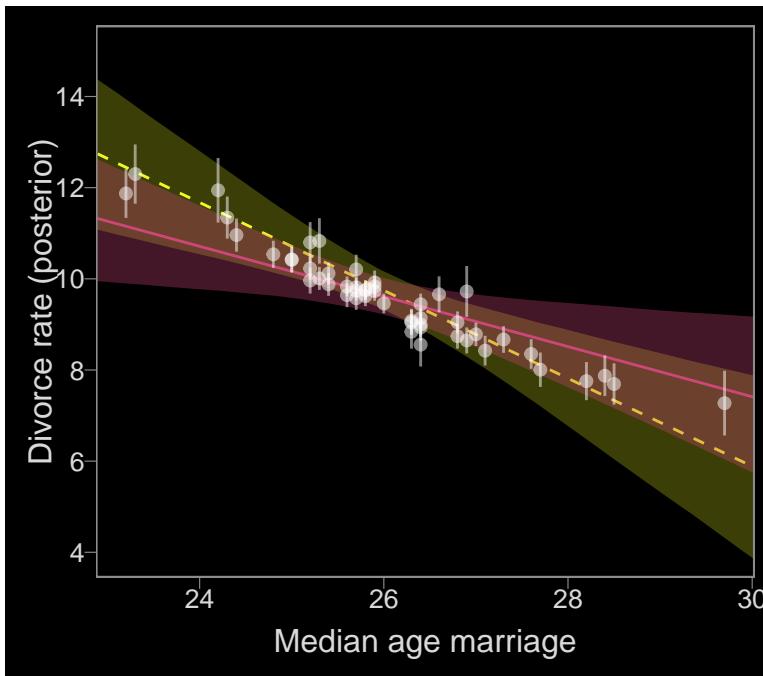
# yellow line
f_no_error <-
  fitted(b14.1b, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

# white dots
data_error <-
  fitted(b14.1_mi) %>%
  as_tibble() %>%
  bind_cols(b14.1_mi$data)

color_y <- viridis_pal(option = "C")(7)[7]
color_r <- viridis_pal(option = "C")(7)[4]

# plot
f_no_error %>%
  ggplot(aes(x = A, y = Estimate)) +
  # `f_no_error`#
  geom_smooth(aes(ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = color_y, color = color_y,
              alpha = 1/4, size = 1/2, linetype = 2) +
  # `f_error`#
  geom_smooth(data = f_error,
              aes(ymin = Q2.5, ymax = Q97.5),
              stat = "identity",
              fill = color_r, color = color_r,
              alpha = 1/3, size = 1/2, linetype = 1) +
  geom_pointrange(data = data_error,
                  aes(ymin = Estimate - Est.Error,
                      ymax = Estimate + Est.Error),
                  color = "white", shape = 20, alpha = 1/2) +
  scale_y_continuous(breaks = seq(from = 4, to = 14, by = 2)) +
  labs(x = "Median age marriage" , y = "Divorce rate (posterior)") +
  coord_cartesian(xlim = range(data_error$A),
                  ylim = c(4, 15))

```



In our plot, it's the reddish regression line that accounts for measurement error.

14.1.2 Error on both outcome and predictor.

In brms, you can specify error on predictors with an `me()` statement in the form of `me(predictor, sd_predictor)` where `sd_predictor` is a vector in the data denoting the size of the measurement error, presumed to be in a standard-deviation metric.

```
# the data
dlist <- list(
  div_obs = d$Divorce,
  div_sd = d$Divorce.SE,
  mar_obs = d$Marriage,
  mar_sd = d$Marriage.SE,
  A       = d$MedianAgeMarriage)

# the `inits`
inits      <- list(Y1 = dlist$div_obs)
inits_list <- list(inits, inits)

# the models
b14.2_se <-
  brm(data = dlist, family = gaussian,
       div_obs | se(div_sd, sigma = TRUE) ~ 0 + intercept + me(mar_obs, mar_sd) + A,
       prior = c(prior(normal(0, 10), class = b),
                 prior(cauchy(0, 2.5), class = sigma)),
       iter = 5000, warmup = 1000, chains = 3, cores = 3,
       seed = 14,
       control = list(adapt_delta = 0.95),
       save_mevars = TRUE) # note the lack if `inits` 

b14.2_mi <-
  brm(data = dlist, family = gaussian,
       div_obs | mi(div_sd) ~ 0 + intercept + me(mar_obs, mar_sd) + A,
       prior = c(prior(normal(0, 10), class = b),
                 prior(cauchy(0, 2.5), class = sigma)),
       iter = 5000, warmup = 1000, cores = 2, chains = 2,
```

```
seed = 14,
control = list(adapt_delta = 0.99,
               max_treedepth = 12),
save_mevars = TRUE,
inits = inits_list)
```

We already know including `inits` values for our `Y1[i]` estimates is a waste of time for our `se()` model. But note how we still defined our `inits` values as `inits <- list(Y1 = dlist$div_obs)` for the `mi()` model. Although it's easy in brms to set the starting values for our `Y1[i]` estimates, much the way McElreath did, that isn't the case when you have measurement error on the predictors. The brms package uses a non-centered parameterization for these, which requires users to have a deeper understanding of the underlying Stan code. This is where I get off the train, but if you want to go further, execute `stancode(b14.2_mi)`.

Here are the two versions of the model.

```
print(b14.2_se)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: div_obs | se(div_sd, sigma = TRUE) ~ 0 + intercept + me(mar_obs, mar_sd) + A
## Data: dlist (Number of observations: 50)
## Samples: 3 chains, each with iter = 5000; warmup = 1000; thin = 1;
##          total post-warmup samples = 12000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## intercept      15.84     6.68    2.50   28.55      4702 1.00
## A              -0.45     0.20   -0.83   -0.05      5401 1.00
## memar_obsmar_sd  0.27     0.10    0.07    0.48      5447 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.00     0.21    0.61    1.44      12592 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

```
print(b14.2_mi)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: div_obs | mi(div_sd) ~ 0 + intercept + me(mar_obs, mar_sd) + A
## Data: dlist (Number of observations: 50)
## Samples: 2 chains, each with iter = 5000; warmup = 1000; thin = 1;
##          total post-warmup samples = 8000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## intercept      15.74     6.54    2.60   28.45      2200 1.00
## A              -0.44     0.20   -0.83   -0.05      2386 1.00
## memar_obsmar_sd  0.27     0.10    0.07    0.48      2189 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma       1.00     0.21    0.62    1.45      1799 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

We'll use `broom::tidy()`, again, to get a sense of `depth=2` summaries.

```
tidy(b14.2_se) %>%
  mutate_if(is.numeric, round, digits = 2)

tidy(b14.2_mi) %>%
  mutate_if(is.numeric, round, digits = 2)
```

Due to space concerns, I'm not going to show the results, here. You can do that on your own. Both methods yielded the posteriors for `Xme_memar_obs[1]`, but only the `b14.2_mi` model based on the `mi()` syntax yielded posteriors for the criterion, the `Y1[i]` summaries.

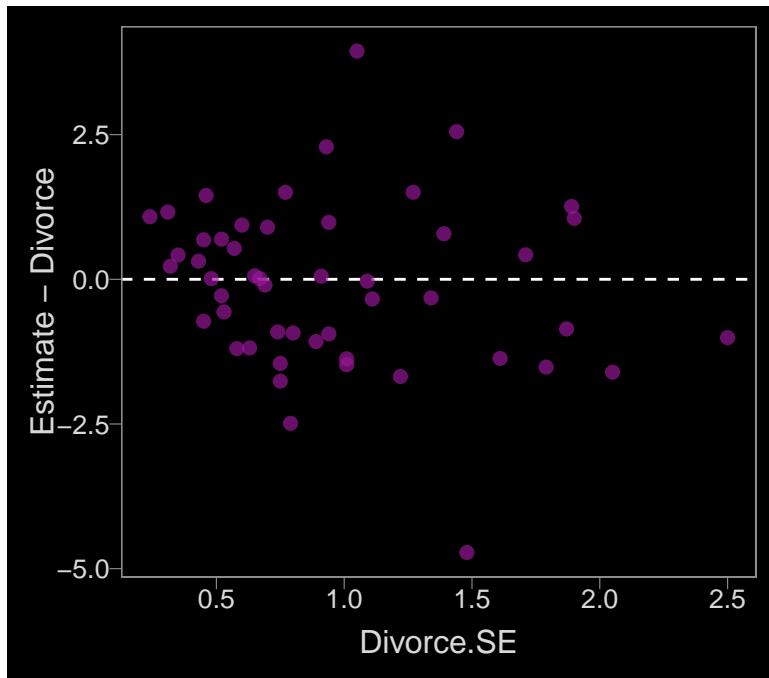
Note that you'll need to specify `save_mevars = TRUE` in the `brm()` function in order to save the posterior samples of error-adjusted variables obtained by using the `me()` argument. Without doing so, functions like `predict()` may give you trouble.

Here is the code for Figure 14.3.a.

```
data_error <-
  fitted(b14.2_mi) %>%
  as_tibble() %>%
  bind_cols(d)

color <- viridis_pal(option = "C")(7)[3]

data_error %>%
  ggplot(aes(x = Divorce.SE, y = Estimate - Divorce)) +
  geom_hline(yintercept = 0, linetype = 2, color = "white") +
  geom_point(alpha = 2/3, size = 2, color = color)
```



To get the posterior samples for error-adjusted `Marriage` rate, we'll use `posterior_samples`. If you examine the object with `glimpse()`, you'll notice 50 `Xme_memar_obs$mar_sd[i]` vectors, with i ranging from 1 to 50, each corresponding to one of the 50 states. With a little data wrangling, you can get the mean of each to put in a plot. Once we have those summaries, we can make our version of Figure 14.4.b.

```
color_y <- viridis_pal(option = "C")(7)[7]
color_p <- viridis_pal(option = "C")(7)[2]

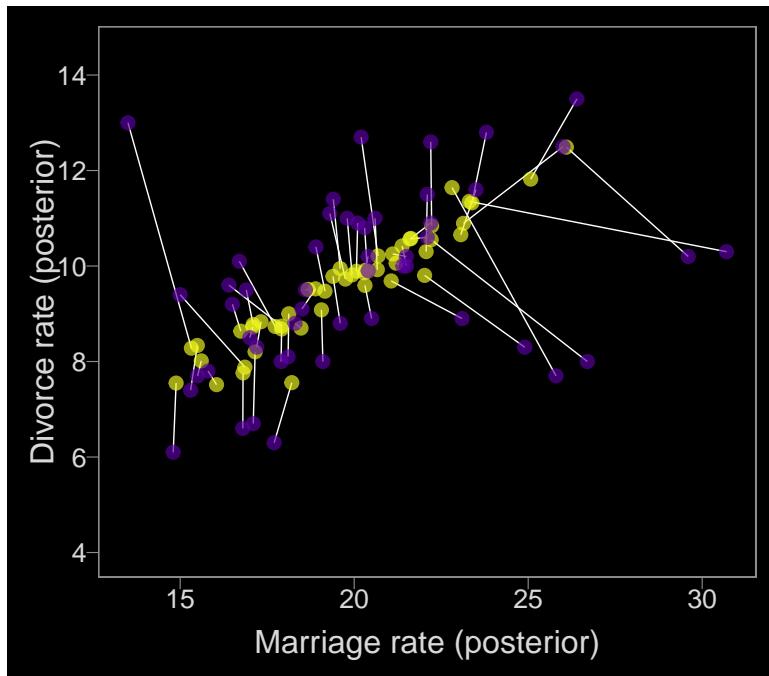
posterior_samples(b14.2_mi) %>%
```

```

select(starts_with("Xme")) %>%
gather() %>%
# this extracts the numerals from the otherwise cumbersome names in `key` and saves them as integers
mutate(key = str_extract(key, "\d+") %>% as.integer()) %>%
group_by(key) %>%
summarise(mean = mean(value)) %>%
bind_cols(data_error) %>%

ggplot(aes(x = mean, y = Estimate)) +
geom_segment(aes(xend = Marriage, yend = Divorce),
             color = "white", size = 1/4) +
geom_point(size = 2, alpha = 2/3, color = color_y) +
geom_point(aes(x = Marriage, y = Divorce),
            size = 2, alpha = 2/3, color = color_p) +
scale_y_continuous(breaks = seq(from = 4, to = 14, by = 2)) +
labs(x = "Marriage rate (posterior)", y = "Divorce rate (posterior)") +
coord_cartesian(ylim = c(4, 14.5))

```



The yellow points are model-implied; the purple ones are of the original data. It turns out our brms model regularized more aggressively than McElreath's rethinking model. I'm unsure of why. If you understand the difference, [please share with the rest of the class](#).

Anyway,

the big take home point for this section is that when you have a distribution of values, don't reduce it down to a single value to use in a regression. Instead, use the entire distribution. Anytime we use an average value, discarding the uncertainty around that average, we risk overconfidence and spurious inference. This doesn't only apply to measurement error, but also to cases which data are averaged before analysis.

Do not average. Instead, model. (p. 431)

14.2 Missing data

Starting with [version 2.2.0](#) brms now supports Bayesian missing data imputation using adaptations of the [multivariate syntax](#). Bürkner's [Handle Missing Values with brms vignette](#) is quite helpful.

14.2.1 Imputing neocortex

Once again, here are the `milk` data.

```
library(rethinking)
data(milk)
d <- milk

d <-
  d %>%
  mutate(neocortex.prop = neocortex.perc / 100,
        logmass      = log(mass))
```

Now we'll switch out `rethinking` for `brms` and do a little data wrangling.

```
detach(package:rethinking, unload = T)
library(brms)
rm(milk)

# prep data
data_list <-
list(
  kcal      = d$kcal.per.g,
  neocortex = d$neocortex.prop,
  logmass   = d$logmass)
```

Here's the structure of our data list.

```
data_list

## $kcal
## [1] 0.49 0.51 0.46 0.48 0.60 0.47 0.56 0.89 0.91 0.92 0.80 0.46 0.71 0.71 0.71 0.73 0.68 0.72 0.97 0.79
## [20] 0.84 0.48 0.62 0.51 0.54 0.49 0.53 0.48 0.55 0.71
##
## $neocortex
## [1] 0.5516     NA     NA     NA     NA 0.6454 0.6454 0.6764     NA 0.6885 0.5885 0.6169 0.6032
## [14]     NA     NA 0.6997     NA 0.7041     NA 0.7340     NA 0.6753     NA 0.7126 0.7260     NA
## [27] 0.7024 0.7630 0.7549
##
## $logmass
## [1] 0.6678294 0.7371641 0.9202828 0.4824261 0.7839015 1.6582281 1.6808279 0.9202828
## [9] -0.3424903 -0.3856625 -2.1202635 -0.7550226 -1.1394343 -0.5108256 1.2441546 0.4382549
## [17] 1.9572739 1.1755733 2.0719133 2.5095993 2.0268316 1.6808279 2.3721112 3.5689692
## [25] 4.3748761 4.5821062 3.7072104 3.4998354 4.0064237
```

Our statistical model follows the form

$$\begin{aligned}
\text{kcal}_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \alpha + \beta_1 \text{neocortex}_i + \beta_2 \text{logmass}_i \\
\text{neocortex}_i &\sim \text{Normal}(\nu, \sigma_{\text{neocortex}}) \\
\alpha &\sim \text{Normal}(0, 100) \\
\beta_1 &\sim \text{Normal}(0, 10) \\
\beta_2 &\sim \text{Normal}(0, 10) \\
\sigma &\sim \text{HalfCauchy}(0, 1) \\
\nu &\sim \text{Normal}(0.5, 1) \\
\sigma_{\text{neocortex}} &\sim \text{HalfCauchy}(0, 1)
\end{aligned}$$

When writing a multivariate model in brms, I find it easier to save the model code by itself and then insert it into the `brm()` function. Otherwise, things get cluttered in a hurry.

```
b_model <-
  # here's the primary `kcal` model
  bf(kcal ~ 1 + mi(neocortex) + logmass) +
  # here's the model for the missing `neocortex` data
  bf(neocortex | mi() ~ 1) +
  # here we set the residual correlations for the two models to zero
  set_rescor(FALSE)
```

Note the `mi(neocortex)` syntax in the `kcal` model. This indicates that the predictor, `neocortex`, has missing values that are themselves being modeled.

To get a sense of how to specify the priors for such a model, use the `get_prior()` function.

```
get_prior(data = data_list,
          family = gaussian,
          b_model)
```

	prior	class	coef	group	resp	dpar	nlpar	bound
## 1		b						
## 2		Intercept						
## 3		b				kcal		
## 4		b	logmass			kcal		
## 5		b	mineocortex			kcal		
## 6	student_t(3, 1, 10)	Intercept				kcal		
## 7	student_t(3, 0, 10)	sigma				kcal		
## 8	student_t(3, 1, 10)	Intercept				neocortex		
## 9	student_t(3, 0, 10)	sigma				neocortex		

With the one-step Bayesian imputation procedure in brms, you might need to use the `resp` argument when specifying non-default priors.

Anyway, here we fit the model.

```
b14.3 <-
  brm(data = data_list,
       family = gaussian,
       b_model, # here we insert the model
       prior = c(prior(normal(0, 100), class = Intercept, resp = kcal),
                 prior(normal(0.5, 1), class = Intercept, resp = neocortex),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 1), class = sigma,     resp = kcal),
                 prior(cauchy(0, 1), class = sigma,     resp = neocortex)),
       iter = 1e4, chains = 2, cores = 2,
       seed = 14)
```

The imputed `neocortex` values are indexed by occasion number from the original data.

```
tidy(b14.3) %>%
  mutate_if(is.numeric, round, digits = 2)
```

	term	estimate	std.error	lower	upper
## 1	b_kcal_Intercept	-0.53	0.47	-1.30	0.25
## 2	b_neocortex_Intercept	0.67	0.01	0.65	0.69
## 3	b_kcal_logmass	-0.07	0.02	-0.11	-0.03
## 4	bsp_kcal_mineocortex	1.89	0.74	0.68	3.09
## 5	sigma_kcal	0.13	0.02	0.10	0.18

```

## 6      sigma_neocortex    0.06    0.01  0.05  0.08
## 7      Ymi_neocortex[2]   0.63    0.05  0.55  0.72
## 8      Ymi_neocortex[3]   0.62    0.05  0.54  0.71
## 9      Ymi_neocortex[4]   0.62    0.05  0.54  0.71
## 10     Ymi_neocortex[5]   0.65    0.05  0.58  0.73
## 11     Ymi_neocortex[9]   0.70    0.05  0.62  0.79
## 12     Ymi_neocortex[14]  0.66    0.05  0.58  0.74
## 13     Ymi_neocortex[15]  0.69    0.05  0.61  0.76
## 14     Ymi_neocortex[17]  0.70    0.05  0.61  0.77
## 15     Ymi_neocortex[19]  0.71    0.05  0.63  0.79
## 16     Ymi_neocortex[21]  0.65    0.05  0.57  0.73
## 17     Ymi_neocortex[23]  0.66    0.05  0.58  0.74
## 18     Ymi_neocortex[26]  0.70    0.05  0.61  0.78
## 19     lp__      40.46   4.36 32.48 46.68

```

Here's the model that drops the cases with NAs on `neocortex`.

```

b14.3cc <-
  brm(data = data_list,
       family = gaussian,
       kcal ~ 1 + neocortex + logmass,
       prior = c(prior(normal(0, 100), class = Intercept),
                  prior(normal(0, 10), class = b),
                  prior(cauchy(0, 1), class = sigma)),
       iter = 1e4, chains = 2, cores = 2,
       seed = 14)

```

The parameters:

```

tidy(b14.3cc) %>%
  mutate_if(is.numeric, round, digits = 2)

```

	term	estimate	std.error	lower	upper
## 1	b_Intercept	-1.07	0.61	-2.04	-0.07
## 2	b_neocortex	2.77	0.95	1.21	4.28
## 3	b_logmass	-0.10	0.03	-0.14	-0.05
## 4	sigma	0.14	0.03	0.10	0.20
## 5	lp__	-4.29	1.71	-7.51	-2.37

In order to make our versions of Figure 14.4, we'll need to do a little data wrangling with `fitted()`.

```

nd <-
  tibble(neocortex = seq(from = .5, to = .85, length.out = 30),
         logmass     = median(data_list$logmass))

f_b14.3 <-
  fitted(b14.3, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

f_b14.3 %>%
  glimpse()

## Observations: 30
## Variables: 10
## $ Estimate.kcal      <dbl> 0.3312196, 0.3540894, 0.3769593, 0.3998291, 0.4226989, 0.4455688, 0...
## $ Est.Error.kcal     <dbl> 0.12585397, 0.11720996, 0.10860587, 0.10005203, 0.09156252, 0.08315...
## $ Q2.5.kcal          <dbl> 0.08679786, 0.12619653, 0.16488722, 0.20378873, 0.24274987, 0.28284...

```

```

## $ Q97.5.kcal      <dbl> 0.5848699, 0.5902992, 0.5960885, 0.6013202, 0.6069732, 0.6133748, 0...
## $ Estimate.neocortex <dbl> 0.6714736, 0.6714736, 0.6714736, 0.6714736, 0.6714736, 0...
## $ Est.Error.neocortex <dbl> 0.01368433, 0.01368433, 0.01368433, 0.01368433, 0.01368433, 0.01368...
## $ Q2.5.neocortex    <dbl> 0.6446126, 0.6446126, 0.6446126, 0.6446126, 0.6446126, 0...
## $ Q97.5.neocortex   <dbl> 0.6980734, 0.6980734, 0.6980734, 0.6980734, 0.6980734, 0...
## $ neocortex         <dbl> 0.5000000, 0.5120690, 0.5241379, 0.5362069, 0.5482759, 0.5603448, 0...
## $ logmass           <dbl> 1.244155, 1.244155, 1.244155, 1.244155, 1.244155, 1.244155...

```

To include the imputed `neocortex` values in the plot, we'll extract the information from `broom::tidy()`.

```

f_b14.3_mi <-  

  tidy(b14.3) %>%  

  filter(str_detect(term, "Ymi")) %>%  

  bind_cols(data_list %>%  

    as_tibble() %>%  

    filter(is.na(neocortex))  

  )  
  

f_b14.3_mi %>% head()

```

	term	estimate	std.error	lower	upper	kcal	neocortex	logmass
## 1	<code>Ymi_neocortex[2]</code>	0.6332440	0.05112357	0.5529586	0.7185007	0.51	NA	0.7371641
## 2	<code>Ymi_neocortex[3]</code>	0.6245501	0.05149849	0.5412463	0.7101070	0.46	NA	0.9202828
## 3	<code>Ymi_neocortex[4]</code>	0.6225438	0.05143366	0.5405524	0.7084048	0.48	NA	0.4824261
## 4	<code>Ymi_neocortex[5]</code>	0.6525390	0.04812462	0.5761665	0.7331180	0.60	NA	0.7839015
## 5	<code>Ymi_neocortex[9]</code>	0.7009097	0.04967649	0.6227489	0.7852700	0.91	NA	-0.3424903
## 6	<code>Ymi_neocortex[14]</code>	0.6567370	0.05031958	0.5766811	0.7398577	0.71	NA	-0.5108256

Data wrangling done—here's our code for Figure 14.4.a.

```

color <- viridis_pal(option = "D")(7)[4]  
  

f_b14.3 %>%  

  ggplot(aes(x = neocortex)) +  

  geom_smooth(aes(y = Estimate.kcal, ymin = Q2.5.kcal, ymax = Q97.5.kcal),  

    stat = "identity",  

    fill = color, color = color, alpha = 1/3, size = 1/2) +  

  geom_point(data = data_list %>% as_tibble(),  

    aes(y = kcal),  

    color = "white") +  

  geom_point(data = f_b14.3_mi,  

    aes(x = estimate, y = kcal),  

    color = color, shape = 1) +  

  geom_segment(data = f_b14.3_mi,  

    aes(x = lower, xend = upper,  

        y = kcal, yend = kcal),  

    color = color, size = 1/4) +  

  coord_cartesian(xlim = c(.55, .8),  

    ylim = range(data_list$kcal, na.rm = T)) +  

  labs(subtitle = "Note: For the regression line in this plot, log(mass)\nhas been set to its median, 1.244.",  

    x = "neocortex proportion",  

    y = "kcal per gram")

```

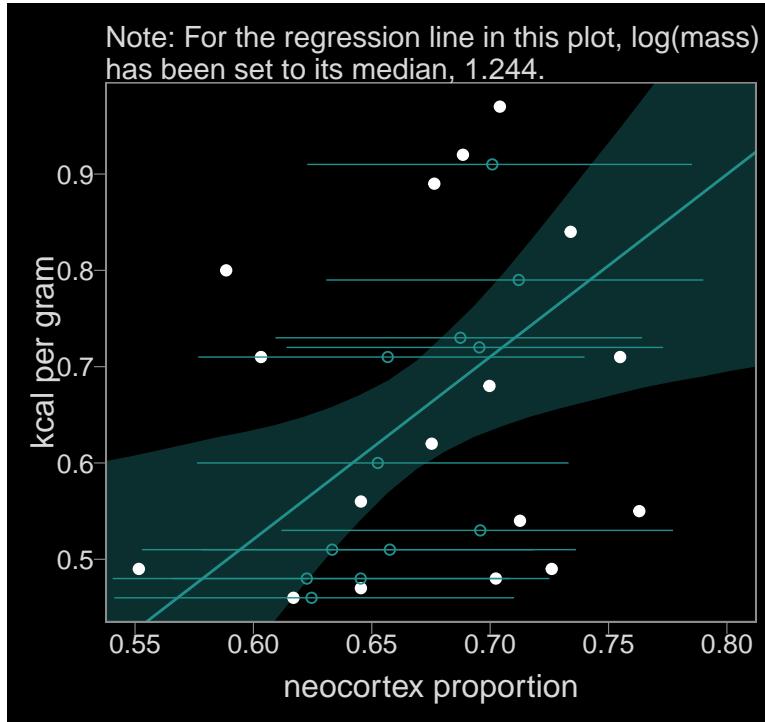


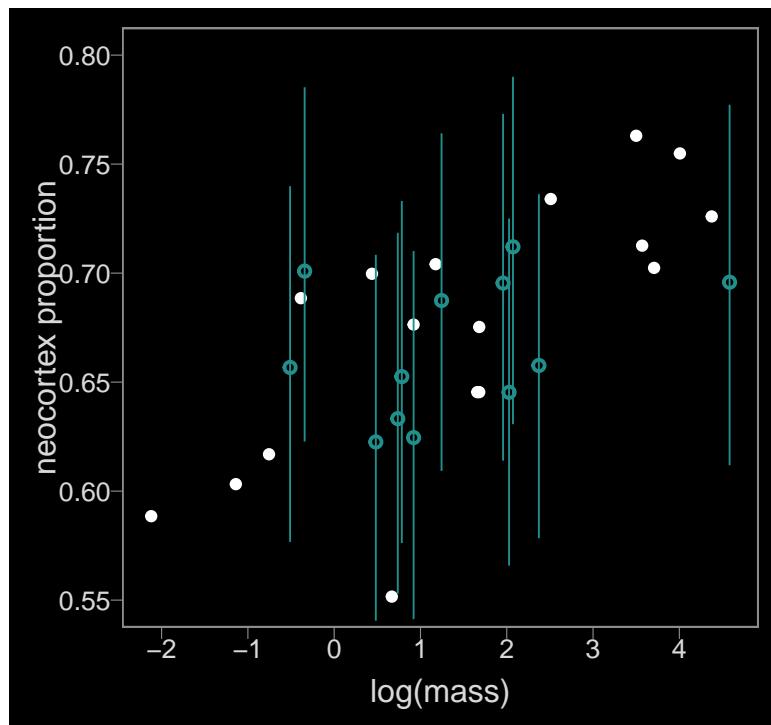
Figure 14.4.b.

```

color <- viridis_pal(option = "D")(7)[4]

data_list %>%
  as_tibble() %>%

ggplot(aes(x = logmass, y = neocortex)) +
  geom_point(color = "white") +
  geom_pointrange(data = f_b14.3_mi,
    aes(y = estimate,
        ymin = lower, ymax = upper),
    color = color, size = 1/3, shape = 1) +
  scale_x_continuous("log(mass)", breaks = -2:4) +
  ylab("neocortex proportion") +
  coord_cartesian(xlim = range(data_list$logmass, na.rm = T),
    ylim = c(.55, .8))
  
```



14.2.2 Improving the imputation model

Like McElreath, we'll update the imputation line of our statistical model to:

$$\text{neocortex}_i \sim \text{Normal}(\nu_i, \sigma_{\text{neocortex}})$$

$$\nu_i = \alpha_{\text{neocortex}} + \gamma_1 \log \text{mass}_i$$

which includes the updated priors

$$\alpha_{\text{neocortex}} \sim \text{Normal}(0.5, 1)$$

$$\gamma_1 \sim \text{Normal}(0, 10)$$

As far as the brms code goes, adding `logmass` as a predictor to the `neocortex` submodel is pretty simple.

```
# define the model
b_model <-
  bf(kcal ~ 1 + mi(neocortex) + logmass) +
  bf(neocortex | mi() ~ 1 + logmass) + # here's the big difference
  set_rescor(FALSE)

# fit the model
b14.4 <-
  brm(data = data_list,
       family = gaussian,
       b_model,
       prior = c(prior(normal(0, 100), class = Intercept, resp = kcal),
                 prior(normal(0.5, 1), class = Intercept, resp = neocortex),
                 prior(normal(0, 10), class = b),
                 prior(cauchy(0, 1), class = sigma,     resp = kcal),
                 prior(cauchy(0, 1), class = sigma,     resp = neocortex)),
       iter = 1e4, chains = 2, cores = 2,
       seed = 14)
```

Behold the parameter estimates.

```
tidy(b14.4) %>%
  mutate_if(is.numeric, round, digits = 2)

## # A tibble: 20 x 6
##   term      estimate std.error lower    upper
##   <chr>     <dbl>     <dbl>    <dbl>   <dbl>
## 1 b_kcal_Intercept -0.88      0.48    -1.64  -0.07
## 2 b_neocortex_Intercept  0.64      0.01     0.62    0.66
## 3 b_kcal_logmass     -0.09      0.02    -0.13   -0.05
## 4 b_neocortex_logmass   0.02      0.01     0.01    0.03
## 5 bsp_kcal_mineocortex  2.46      0.75     1.19    3.65
## 6 sigma_kcal        0.13      0.02     0.10    0.17
## 7 sigma_neocortex     0.04      0.01     0.03    0.06
## 8 Ymi_neocortex[2]     0.63      0.03     0.57    0.69
## 9 Ymi_neocortex[3]     0.63      0.04     0.57    0.69
## 10 Ymi_neocortex[4]     0.62      0.04     0.56    0.68
## 11 Ymi_neocortex[5]     0.65      0.03     0.59    0.70
## 12 Ymi_neocortex[9]     0.66      0.04     0.60    0.72
## 13 Ymi_neocortex[14]     0.63      0.03     0.57    0.68
## 14 Ymi_neocortex[15]     0.68      0.03     0.62    0.74
## 15 Ymi_neocortex[17]     0.70      0.03     0.64    0.75
## 16 Ymi_neocortex[19]     0.71      0.03     0.65    0.77
## 17 Ymi_neocortex[21]     0.66      0.03     0.61    0.72
## 18 Ymi_neocortex[23]     0.68      0.03     0.62    0.73
## 19 Ymi_neocortex[26]     0.74      0.04     0.68    0.80
## 20 lp__                48.90     4.05    41.43   54.75
```

Here's our pre-Figure 14.5 data wrangling.

```
f_b14.4 <-
  fitted(b14.4, newdata = nd) %>%
  as_tibble() %>%
  bind_cols(nd)

f_b14.4_mi <-
  tidy(b14.4) %>%
  filter(str_detect(term, "Ymi")) %>%
  bind_cols(data_list %>%
    as_tibble() %>%
    filter(is.na(neocortex)))
  )

f_b14.4 %>%
  glimpse()

## # Observations: 30
## # Variables: 10
## # $ Estimate.kcal      <dbl> 0.2380801, 0.2677612, 0.2974423, 0.3271235, 0.3568046, 0.3864857, 0...
## # $ Est.Error.kcal     <dbl> 0.12785556, 0.11906079, 0.11030292, 0.10159151, 0.09293962, 0.08436...
## # $ Q2.5.kcal          <dbl> -0.009206759, 0.037348106, 0.083427138, 0.129856800, 0.176586987, 0...
## # $ Q97.5.kcal         <dbl> 0.5025289, 0.5142660, 0.5262648, 0.5377310, 0.5487249, 0.5612721, 0...
## # $ Estimate.neocortex <dbl> 0.6670467, 0.6670467, 0.6670467, 0.6670467, 0.6670467, 0.6670467, 0...
## # $ Est.Error.neocortex <dbl> 0.009622526, 0.009622526, 0.009622526, 0.009622526, 0.009622526, 0....
## # $ Q2.5.neocortex      <dbl> 0.6478488, 0.6478488, 0.6478488, 0.6478488, 0.6478488, 0.6478488, 0...
## # $ Q97.5.neocortex     <dbl> 0.685581, 0.685581, 0.685581, 0.685581, 0.685581, 0.685581, 0.68558...
## # $ neocortex            <dbl> 0.5000000, 0.5120690, 0.5241379, 0.5362069, 0.5482759, 0.5603448, 0...
## # $ logmass              <dbl> 1.244155, 1.244155, 1.244155, 1.244155, 1.244155, 1.244155, 1.24415...
```

```
f_b14.4_mi %>%
  glimpse()
```

```
## Observations: 12
## Variables: 8
## $ term      <chr> "Ymi_neocortex[2]", "Ymi_neocortex[3]", "Ymi_neocortex[4]", "Ymi_neocortex[5]...
## $ estimate   <dbl> 0.6310163, 0.6284698, 0.6195025, 0.6463966, 0.6629660, 0.6273373, 0.6796101, ...
## $ std.error  <dbl> 0.03443509, 0.03515450, 0.03533123, 0.03347496, 0.03580270, 0.03456625, 0.034...
## $ lower     <dbl> 0.5741997, 0.5712126, 0.5617213, 0.5924678, 0.6040286, 0.5711475, 0.6242119, ...
## $ upper     <dbl> 0.6875889, 0.6862857, 0.6765639, 0.7024188, 0.7213913, 0.6839601, 0.7362796, ...
## $ kcal       <dbl> 0.51, 0.46, 0.48, 0.60, 0.91, 0.71, 0.73, 0.72, 0.79, 0.48, 0.51, 0.53
## $ neocortex <dbl> NA, NA
## $ logmass    <dbl> 0.7371641, 0.9202828, 0.4824261, 0.7839015, -0.3424903, -0.5108256, 1.2441546...
```

For our final plots, let's play around with colors from `viridis_pal(option = "D")`. Figure 14.5.a.

```
color <- viridis_pal(option = "D")(7)[3]

f_b14.4 %>%
  ggplot(aes(x = neocortex)) +
  geom_smooth(aes(y = Estimate.kcal, ymin = Q2.5.kcal, ymax = Q97.5.kcal),
              stat = "identity",
              fill = color, color = color, alpha = 1/2, size = 1/2) +
  geom_point(data = data_list %>% as_tibble(),
             aes(y = kcal),
             color = "white") +
  geom_point(data = f_b14.4_mi,
             aes(x = estimate, y = kcal),
             color = color, shape = 1) +
  geom_segment(data = f_b14.4_mi,
               aes(x = lower, xend = upper,
                   y = kcal, yend = kcal),
               color = color, size = 1/4) +
  coord_cartesian(xlim = c(.55, .8),
                  ylim = range(data_list$kcal, na.rm = T)) +
  labs(subtitle = "Note: For the regression line in this plot, log(mass)\nhas been set to its median, 1.244.",
       x = "neocortex proportion",
       y = "kcal per gram")
```

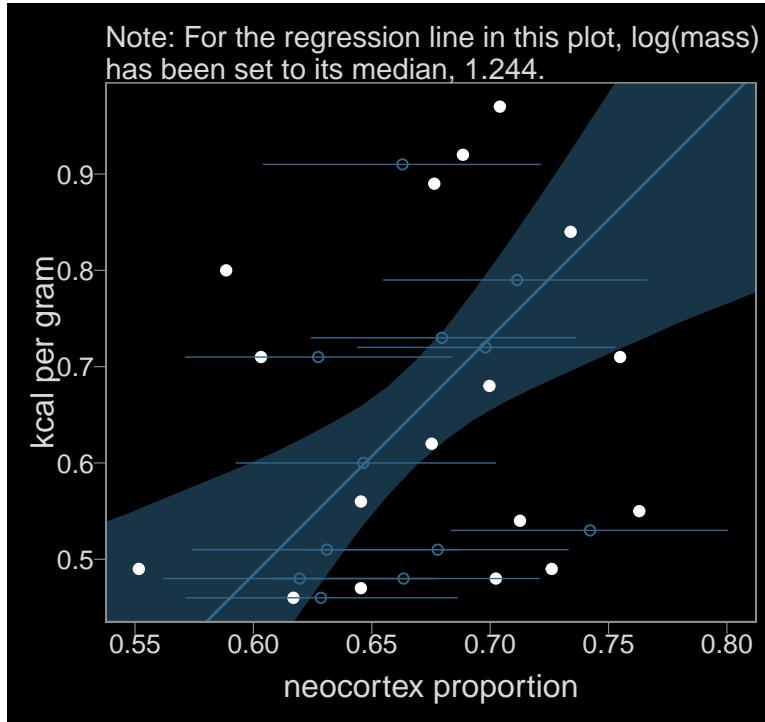


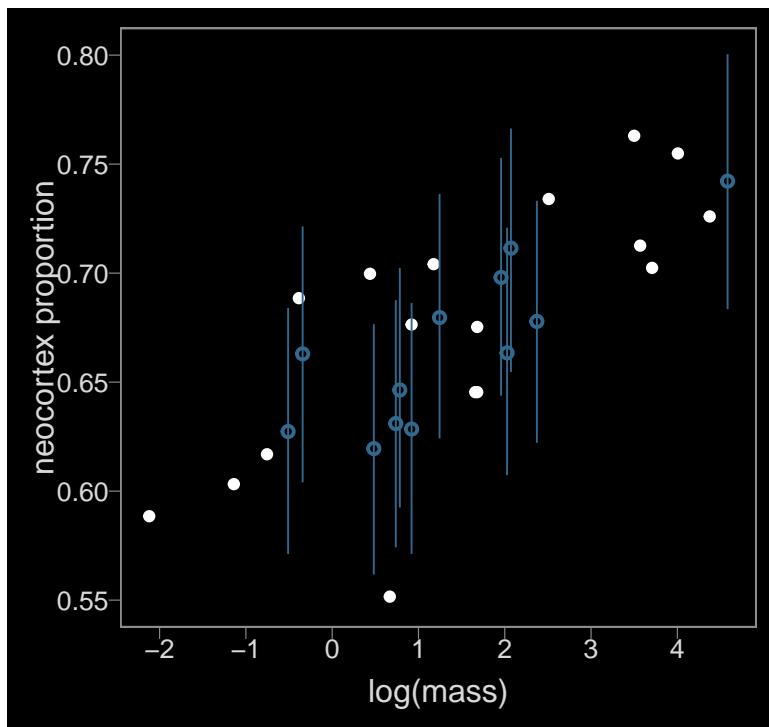
Figure 14.5.b.

```

color <- viridis_pal(option = "D")(7)[3]

data_list %>%
  as_tibble() %>%

ggplot(aes(x = logmass, y = neocortex)) +
  geom_point(color = "white") +
  geom_pointrange(data = f_b14.4_mi,
    aes(y = estimate,
        ymin = lower, ymax = upper),
    color = color, size = 1/3, shape = 1) +
  scale_x_continuous("log(mass)", breaks = -2:4) +
  ylab("neocortex proportion") +
  coord_cartesian(xlim = range(data_list$logmass, na.rm = T),
    ylim = c(.55, .8))
  
```



If modern missing data methods are new to you, you might also check out van Buuren's great online text [Flexible Imputation of Missing Data. Second Edition](#). I'm also a fan of Enders' [Applied Missing Data Analysis](#), for which you can find a free sample chapter [here](#). I'll also quickly mention that `brms` accommodates multiple imputation, too.

14.3 Summary Bonus: Meta-analysis

If your mind isn't fully blown by those measurement-error and missing-data models, let's keep building. As it turns out, meta-analyses are often just special kinds of multilevel measurement-error models. Thus, you can use `brms::brm()` to fit Bayesian meta-analyses, too.

Before we proceed, I should acknowledge that this section is heavily influenced by Matti Vuorre's great blog post, [Meta-analysis is a special case of Bayesian multilevel modeling](#). And since McElreath's text doesn't directly address meta-analyses, we'll also have to borrow a bit from Gelman, Carlin, Stern, Dunson, Vehtari, and Rubin's [Bayesian data analysis, Third edition](#). We'll let Gelman and colleagues introduce the topic:

Discussions of meta-analysis are sometimes imprecise about the estimands of interest in the analysis, especially when the primary focus is on testing the null hypothesis of no effect in any of the studies to be combined. Our focus is on estimating meaningful parameters, and for this objective there appear to be three possibilities, accepting the overarching assumption that the studies are comparable in some broad sense. The first possibility is that we view the studies as identical replications of each other, in the sense we regard the individuals in all the studies as independent samples from a common population, with the same outcome measures and so on. A second possibility is that the studies are so different that the results of any one study provide no information about the results of any of the others. A third, more general, possibility is that we regard the studies as exchangeable but not necessarily either identical or completely unrelated; in other words we allow differences from study to study, but such that the differences are not expected *a priori* to have predictable effects favoring one study over another.... this third possibility represents a continuum between the two extremes, and it is this exchangeable model (with unknown hyperparameters characterizing the population distribution) that forms the basis of our Bayesian analysis...

The first potential estimand of a meta-analysis, or a hierarchically structured problem in general, is the mean of the distribution of effect sizes, since this represents the overall 'average' effect across all studies that could be regarded as exchangeable with the observed studies. Other possible estimands are the effect size in any of the observed studies and the effect size in another, comparable (exchangeable) unobserved study. (pp. 125–126, *emphasis* in the original)

The basic version of a Bayesian meta-analysis follows the form

$$y_i \sim \text{Normal}(\theta_i, \sigma_i)$$

where y_i = the point estimate for the effect size of a single study, i , which is presumed to have been a draw from a Normal distribution centered on θ_i . The data in meta-analyses are typically statistical summaries from individual studies. The one clear lesson from this chapter is that those estimates themselves come with error and those errors should be fully expressed in the meta-analytic model. Which we do. The standard error from study i is specified σ_i , which is also a stand-in for the standard deviation of the Normal distribution from which the point estimate was drawn. Do note, we're not estimating σ_i , here. Those values we take directly from the original studies.

Building on the model, we further presume that study i is itself just one draw from a population of related studies, each of which have their own effect sizes. As such, we presume θ_i itself has a distribution following the form

$$\theta_i \sim \text{Normal}(\mu, \tau)$$

where μ is the meta-analytic effect (i.e., the population mean) and τ is the variation around that mean, what you might also think of as σ_τ .

Since there's no example of a meta-analysis in the text, we'll have to get our data elsewhere. We'll focus on Gershoff and Grogan-Kaylor's (2016) paper, *Spanking and Child Outcomes: Old Controversies and New Meta-Analyses*. From their introduction, we read:

Around the world, most children (80%) are spanked or otherwise physically punished by their parents (UNICEF, 2014). The question of whether parents should spank their children to correct misbehaviors sits at a nexus of arguments from ethical, religious, and human rights perspectives both in the U.S. and around the world (Gershoff, 2013). Several hundred studies have been conducted on the associations between parents' use of spanking or physical punishment and children's behavioral, emotional, cognitive, and physical outcomes, making spanking one of the most studied aspects of parenting. What has been learned from these hundreds of studies? (p. 453)

Our goal will be to learn Bayesian meta-analysis by answering part of that question. I've transcribed the values directly from Gershoff and Grogan-Kaylor's paper and saved them as a file called `spank.xlsx`. You can find the data in [this project's GitHub repository](#). Let's load them and `glimpse()`.

```
spank <- readxl::read_excel("spank.xlsx")

glimpse(spank)

## #> #> Observations: 111
## #> #> Variables: 8
## #> #> $ study    <chr> "Bean and Roberts (1981)", "Day and Roberts (1983)", "Minton, Kagan, and Levine...
## #> #> $ year     <dbl> 1981, 1983, 1971, 1988, 1990, 1961, 1962, 1990, 2002, 2005, 1986, 2012, 1979, 2...
## #> #> $ outcome   <chr> "Immediate defiance", "Immediate defiance", "Immediate defiance", "Immediate de...
## #> #> $ between   <dbl> 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0...
## #> #> $ within    <dbl> 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1...
## #> #> $ d         <dbl> -0.74, 0.36, 0.34, -0.08, 0.10, 0.63, 0.19, 0.47, 0.14, -0.18, 1.18, 0.70, 0.63...
## #> #> $ ll        <dbl> -1.76, -1.04, -0.09, -1.01, -0.82, 0.16, -0.14, 0.20, -0.42, -0.49, 0.15, 0.35...
## #> #> $ ul        <dbl> 0.28, 1.77, 0.76, 0.84, 1.03, 1.10, 0.53, 0.74, 0.70, 0.13, 2.22, 1.05, 1.71, 0...
```

In this paper, the effect size of interest is a *Cohen's d*, derived from the formula

$$d = \frac{\mu_{\text{treatment}} - \mu_{\text{comparison}}}{\sigma_{\text{pooled}}}$$

where

$$\sigma_{\text{pooled}} = \sqrt{\frac{(n_1 - 1)\sigma_1^2 + ((n_2 - 1)\sigma_2^2)}{n_1 + n_2 - 2}}$$

To help make the equation for d clearer for our example, we might re-express it as

$$d = \frac{\mu_{\text{spanked}} - \mu_{\text{not spanked}}}{\sigma_{\text{pooled}}}$$

McElreath didn't really focus on effect sizes in his text. If you need a refresher, you might check out Kelley and Preacher's [On effect size](#). But in words, *Cohen's d* is a standardized mean difference between two groups.

So if you look back up at the results of `glimpse(spank)` you'll notice the column `d`, which is indeed a vector of *Cohen's d* effect sizes. The last two columns, `ll` and `ul`, are the lower and upper limits of the associated 95% frequentist confidence intervals. But we don't want confidence intervals for our `d`-values; we want their standard errors. Fortunately, we can compute those with the following formula

$$SE = \frac{\text{upper limit} - \text{lower limit}}{3.92}$$

Here it is in code.

```
spank <-
  spank %>%
  mutate(se = (ul - ll) / 3.92)

glimpse(spank)
```

```
## Observations: 111
## Variables: 9
## $ study    <chr> "Bean and Roberts (1981)", "Day and Roberts (1983)", "Minton, Kagan, and Levine...
## $ year     <dbl> 1981, 1983, 1971, 1988, 1990, 1961, 1962, 1990, 2002, 2005, 1986, 2012, 1979, 2...
## $ outcome   <chr> "Immediate defiance", "Immediate defiance", "Immediate defiance", "Immediate de...
## $ between   <dbl> 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0...
## $ within    <dbl> 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1...
## $ d         <dbl> -0.74, 0.36, 0.34, -0.08, 0.10, 0.63, 0.19, 0.47, 0.14, -0.18, 1.18, 0.70, 0.63...
## $ ll        <dbl> -1.76, -1.04, -0.09, -1.01, -0.82, 0.16, -0.14, 0.20, -0.42, -0.49, 0.15, 0.35, ...
## $ ul        <dbl> 0.28, 1.77, 0.76, 0.84, 1.03, 1.10, 0.53, 0.74, 0.70, 0.13, 2.22, 1.05, 1.71, 0...
## $ se        <dbl> 0.52040816, 0.71683673, 0.21683673, 0.47193878, 0.47193878, 0.23979592, 0.17091...
```

Now our data are ready, we can express our first Bayesian meta-analysis with the formula

$$\begin{aligned} d_i &\sim \text{Normal}(\theta_i, \sigma_i = \text{se}_i) \\ \theta_i &\sim \text{Normal}(\mu, \tau) \\ \mu &\sim \text{Normal}(0, 1) \\ \tau &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

The last two lines, of course, spell out our priors. In psychology, it's pretty rare to see *Cohen's d*-values greater than the absolute value of ± 1 . So in the absence of more specific domain knowledge—which I don't have—, it seems like $\text{Normal}(0, 1)$ is a reasonable place to start. And just like McElreath used $\text{HalfCauchy}(0, 1)$ as the default prior for the group-level standard deviations, [it makes sense to use it here](#) for our meta-analytic τ parameter.

Here's the code for the first model.

```
b14.5 <-
  brm(data = spank, family = gaussian,
       d | se ~ 1 + (1 | study),
       prior = c(prior(normal(0, 1), class = Intercept),
                 prior(cauchy(0, 1), class = sd)),
       iter = 4000, warmup = 1000, cores = 4, chains = 4,
       seed = 14)
```

One thing you might notice is our `se(se)` function excluded the `sigma` argument. If you recall from section 14.1, we specified `sigma = T` in our measurement-error models. The `brms` default is that within `se()`, `sigma = FALSE`. As such, we have no estimate for `sigma` the way we would if we were doing this analysis with the raw data from the studies. Hopefully this makes sense. The uncertainty around the d -value for each study i has already been encoded in the data as `se`.

This brings us to another point. We typically perform meta-analyses on data summaries. In my field and perhaps in yours, this is due to the historical accident that it has not been the norm among researchers to make their data publically available. So effect size summaries were the best we typically had. However, times are changing (e.g., [here](#), [here](#)). If the raw data from all the studies for your meta-analysis are available, you can just fit a multilevel model in which the data are nested in the studies. Heck, you could even allow the studies to vary by σ by taking the [distributional modeling approach](#) and specify something like `sigma ~ 0 + study` or even `sigma ~ 1 + (1 | study)`. But enough technical talk. Let's look at the model results.

```
print(b14.5)
```

```
## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: d | se(se) ~ 1 + (1 | study)
## Data: spank (Number of observations: 111)
## Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 1;
##           total post-warmup samples = 12000
##
## Group-Level Effects:
##   ~study (Number of levels: 76)
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
##   sd(Intercept)    0.26      0.03     0.21     0.33        2411 1.00
## 
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
##   Intercept       0.38      0.04     0.30     0.45        1330 1.00
## 
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Thus, in our simple Bayesian meta-analysis, we have a population *Cohen's d* of about 0.38. Our estimate for τ , 0.26, suggests we have quite a bit of between-study variability. One question you might ask is: *What exactly are these Cohen's ds measuring, anyways?* We've encoded that in the `outcome` vector of the `spank` data.

```
spank %>%
  distinct(outcome) %>%
  knitr::kable()
```

outcome
Immediate defiance
Low moral internalization
Child aggression
Child antisocial behavior
Child externalizing behavior problems
Child internalizing behavior problems
Child mental health problems
Child alcohol or substance abuse
Negative parent-child relationship
Impaired cognitive ability
Low self-esteem
Low self-regulation
Victim of physical abuse
Adult antisocial behavior
Adult mental health problems
Adult alcohol or substance abuse
Adult support for physical punishment

There are a few things to note. First, with the possible exception of `Adult support for physical punishment`, all of the outcomes are negative. We prefer conditions associated with lower values for things like `Child aggression` and `Adult mental health problems`. Second, the way the data are coded, larger effect sizes are interpreted as more negative outcomes associated with children having been spanked. That is, our analysis suggests spanking children is associated with worse outcomes. What might not be immediately apparent is that even though there are 111 cases in the data, there are only 76 distinct studies.

```
spank %>%
  distinct(study) %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     76
```

In other words, some studies have multiple outcomes. In order to better accommodate the `study`- and `outcome`-level variances, let's fit a cross-classified Bayesian meta-analysis reminiscent of the cross-classified chimp model from Chapter 13.

```
b14.6 <- brm(data = spank, family = gaussian,
  d | se(se) ~ 1 + (1 | study) + (1 | outcome),
  prior = c(prior(normal(0, 1), class = Intercept),
            prior(cauchy(0, 1), class = sd)),
  iter = 4000, warmup = 1000, cores = 4, chains = 4,
  seed = 14)

print(b14.6)

## Family: gaussian
##   Links: mu = identity; sigma = identity
## Formula: d | se(se) ~ 1 + (1 | study) + (1 | outcome)
## Data: spank (Number of observations: 111)
## Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 1;
##           total post-warmup samples = 12000
##
## Group-Level Effects:
##   ~outcome (Number of levels: 17)
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
##   sd(Intercept)    0.08      0.03     0.04     0.14        2797 1.00
##   ~study (Number of levels: 76)
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
##   sd(Intercept)    0.25      0.03     0.20     0.32        2277 1.00
##
## Population-Level Effects:
##             Estimate Est.Error 1-95% CI u-95% CI Eff.Sample Rhat
##   Intercept      0.36      0.04     0.28     0.44        1824 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Now we have two τ parameters. We might plot them to get a sense of where the variance is at.

```
# we'll want this to label the plot
label <-
  tibble(tau = c(.12, .3),
```

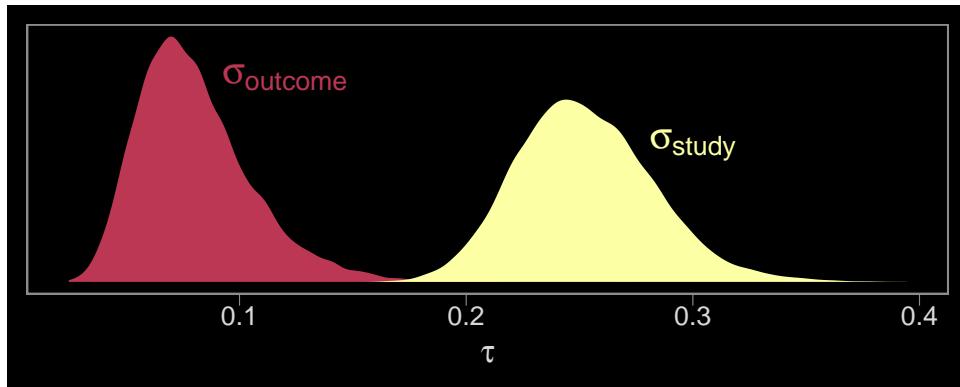
```

y      = c(15, 10),
label = c("sigma['outcome']", "sigma['study']"))

# wrangle
posterior_samples(b14.6) %>%
  select(starts_with("sd")) %>%
  gather(key, tau) %>%
  mutate(key = str_remove(key, "sd_") %>% str_remove(., "__Intercept")) %>%

# plot
ggplot(aes(x = tau)) +
  geom_density(aes(fill = key),
               color = "transparent") +
  geom_text(data = label,
            aes(y = y, label = label, color = label),
            parse = T, size = 5) +
  scale_fill_viridis_d(NULL, option = "B", begin = .5) +
  scale_color_viridis_d(NULL, option = "B", begin = .5) +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab(expression(tau)) +
  theme(panel.grid = element_blank())

```



So at this point, the big story is there's more variability between the studies than there is the outcomes. But I still want to get a sense of the individual outcomes. Here we'll use `tidybayes::geom_halfeyeh()` to help us make our version of a `forest plot` and `tidybayes::spread_draws()` to help with the initial wrangling.

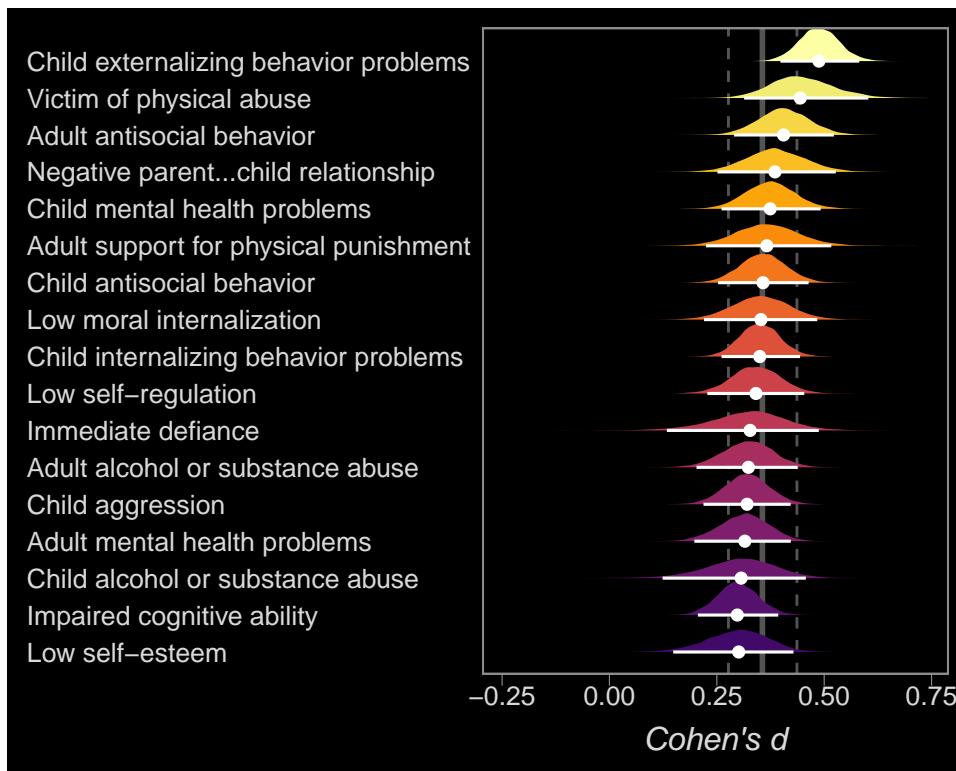
```

library(tidybayes)

b14.6 %>%
  spread_draws(b_Intercept, r_outcome[outcome,]) %>%
  # add the grand mean to the group-specific deviations
  mutate(mu = b_Intercept + r_outcome) %>%
  ungroup() %>%
  mutate(outcome = str_replace_all(outcome, "[.]", " "))

# plot
ggplot(aes(x = mu, y = reorder(outcome, mu), fill = reorder(outcome, mu))) +
  geom_vline(xintercept = fixef(b14.6)[1, 1], color = "grey33", size = 1) +
  geom_vline(xintercept = fixef(b14.6)[1, 3:4], color = "grey33", linetype = 2) +
  geom_halfeyeh(.width = .95, size = 2/3, color = "white") +
  scale_fill_viridis_d(option = "B", begin = .2) +
  labs(x = expression(italic("Cohen's d")),
       y = NULL) +
  theme(panel.grid    = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y  = element_text(hjust = 0))

```



The solid and dashed vertical white lines in the background mark off the grand mean (i.e., the meta-analytic effect) and its 95% intervals. But anyway, there's not a lot of variability across the outcomes. Let's go one step further with the model. Doubling back to Gelman and colleagues, we read:

When assuming exchangeability we assume there are no important covariates that might form the basis of a more complex model, and this assumption (perhaps misguidedly) is widely adopted in meta-analysis. What if other information (in addition to the data (n, y)) is available to distinguish among the J studies in a meta-analysis, so that an exchangeable model is inappropriate? In this situation, we can expand the framework of the model to be exchangeable in the observed data and covariates, for example using a hierarchical regression model. (p. 126)

One important covariate Gershoff and Grogan-Kaylor addressed in their meta-analysis was the type of study. The 76 papers they based their meta-analysis on contained both between- and within-participants designs. In the `spank` data, we've dummy coded that information with the `between` and `within` vectors. Both are dummy variables and `within = 1 - between`. Here are the counts.

```
spank %>%
  count(between)
```

```
## # A tibble: 2 x 2
##   between     n
##   <dbl> <int>
## 1     0     71
## 2     1     40
```

When I use dummies in my models, I prefer to have the majority group stand as the reference category. As such, I typically name those variables by the minority group. In this case, most occasions are based on within-participant designs. Thus, we'll go ahead and add the `between` variable to the model. While we're at it, we'll practice using the `0 + intercept` syntax.

```
b14.7 <- brm(data = spank, family = gaussian,
  d | se(se) ~ 0 + intercept + between + (1 | study) + (1 | outcome),
  prior = c(prior(normal(0, 1), class = b),
            prior(cauchy(0, 1), class = sd)),
  iter = 4000, warmup = 1000, cores = 4, chains = 4,
  seed = 14)
```

Behold the summary.

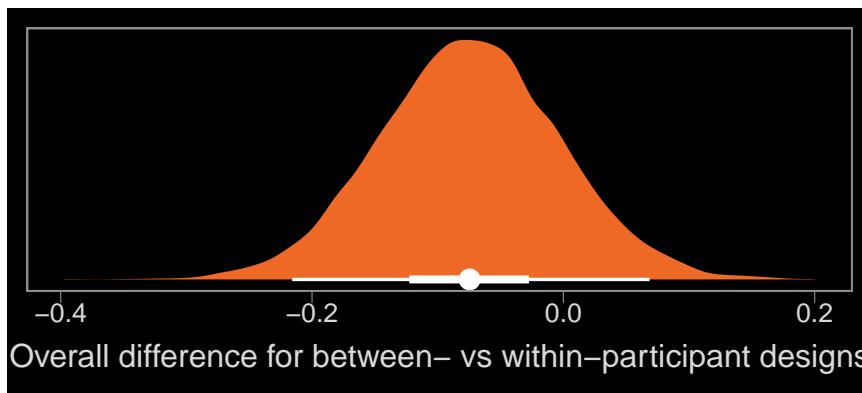
```
print(b14.7)

## Family: gaussian
## Links: mu = identity; sigma = identity
## Formula: d | se(se) ~ 0 + intercept + between + (1 | study) + (1 | outcome)
## Data: spank (Number of observations: 111)
## Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 1;
##          total post-warmup samples = 12000
##
## Group-Level Effects:
## ~outcome (Number of levels: 17)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)     0.08      0.02      0.04      0.14        4758 1.00
##
## ~study (Number of levels: 76)
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sd(Intercept)     0.25      0.03      0.20      0.32        3789 1.00
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## intercept       0.38      0.05      0.29      0.48        2987 1.00
## between         -0.07      0.07     -0.22      0.07        3118 1.00
##
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
## is a crude measure of effective sample size, and Rhat is the potential
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

Let's take a closer look at `b_between`.

```
color <- viridis_pal(option = "B")(7)[5]

posterior_samples(b14.7) %>%
  ggplot(aes(x = b_between, y = 0)) +
  geom_halfeyeh(.width = c(.5, .95), color = "white", fill = color) +
  scale_y_continuous(NULL, breaks = NULL) +
  xlab("Overall difference for between- vs within-participant designs") +
  theme(panel.grid = element_blank())
```



That difference isn't as large I'd expect it to be. But then again, I'm no spanking researcher. So what do I know?

There are other things you might do with these data. For example, you might check for trends by year or, as the authors did in their manuscript, distinguish among different severities of corporal punishment. But I think we've gone far enough to get you started.

If you'd like to learn more about these methods, do check out Vourré's [Meta-analysis is a special case of Bayesian multilevel modeling](#). From his blog, you'll learn additional tricks, like making a more traditional-looking forest plot with the `brmstools::forest()` function and how our Bayesian brms method compares with Frequentist meta-analyses via the `metafor package`. You might also check out Williams, Rast, and Bürkner's manuscript, [Bayesian Meta-Analysis with Weakly Informative Prior Distributions](#) to give you an empirical justification for using a half-Cauchy prior for your meta-analysis τ parameters.

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()

## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel stats      graphics grDevices utils      datasets methods  base
##
## other attached packages:
## [1] tidybayes_1.0.4     broom_0.5.1          viridis_0.5.1       viridisLite_0.3.0
## [5] brms_2.8.8          Rcpp_1.0.1           rstan_2.18.2        StanHeaders_2.18.0-1
## [9]forcats_0.3.0       stringr_1.4.0        dplyr_0.8.0.1       purrr_0.2.5
## [13]readr_1.1.1         tidyr_0.8.1          tibble_2.1.1        ggplot2_3.1.1
## [17]tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
## [1] colorspace_1.3-2    ggridges_0.5.0       rsconnect_0.8.8
## [4] rprojroot_1.3-2    ggstance_0.3         markdown_0.8
## [7] base64enc_0.1-3    rstudioapi_0.7       svUnit_0.7-12
## [10]DT_0.4              fansi_0.4.0          mvtnorm_1.0-10
## [13]lubridate_1.7.4    xml2_1.2.0           bridgesampling_0.6-0
## [16]knitr_1.20           shinythemes_1.1.1    bayesplot_1.6.0
## [19]jsonlite_1.5         shiny_1.1.0           compiler_3.5.1
## [22]httr_1.3.1          backports_1.1.4      assertthat_0.2.0
## [25]Matrix_1.2-14        lazyeval_0.2.2       cli_1.0.1
## [28]later_0.7.3          htmltools_0.3.6      prettyunits_1.0.2
## [31]tools_3.5.1          igraph_1.2.1          coda_0.19-2
## [34]gtable_0.3.0          glue_1.3.1.9000     reshape2_1.4.3
## [37]cellranger_1.1.0    nlme_3.1-137          crosstalk_1.0.0
## [40]xfun_0.3              ps_1.2.1             rvest_0.3.2
## [43]mime_0.5              miniUI_0.1.1.1       gtools_3.8.1
## [46]MASS_7.3-50            zoo_1.8-2            scales_1.0.0
## [49]colourpicker_1.0      hms_0.4.2             promises_1.0.1
## [52]Broddingnag_1.2-6    inline_0.3.15        shinystan_2.5.0
```

```
## [55] yaml_2.1.19          gridExtra_2.3           loo_2.1.0
## [58] stringi_1.4.3         dygraphs_1.1.1.5        pkgbuild_1.0.2
## [61] rlang_0.3.4           pkgconfig_2.0.2         matrixStats_0.54.0
## [64] evaluate_0.10.1       lattice_0.20-35        rstantools_1.5.1
## [67] htmlwidgets_1.2        labeling_0.3            processx_3.2.1
## [70] tidyselect_0.2.5       plyr_1.8.4              magrittr_1.5
## [73] bookdown_0.9          R6_2.3.0                generics_0.0.2
## [76] pillar_1.3.1          haven_1.1.2             withr_2.1.2
## [79] xts_0.10-2            abind_1.4-5             modelr_0.1.2
## [82] crayon_1.3.4          arrayhelpers_1.0-20160527 utf8_1.1.4
## [85] rmarkdown_1.10          grid_3.5.1               readxl_1.1.0
## [88] callr_3.1.0           threejs_0.3.1            digest_0.6.18
## [91] xtable_1.8-2          httpuv_1.4.4.2          stats4_3.5.1
## [94] munsell_0.5.0          shinyjs_1.0
```


Chapter 15

Horoscopes Insights

Statistical inference is indeed critically important. But only as much as every other part of research. Scientific discovery is not an additive process, in which sin in one part can be atoned by virtue in another. Everything interacts. So equally when science works as intended as when it does not, every part of the process deserves attention. (p. 441)

In this final chapter, there are no models for us to fit and no figures for use to reimagine. McElreath took the opportunity to comment more broadly on the scientific process. He made a handful of great points, some of which I'll quote in a bit. But for the bulk of this chapter, I'd like to take the opportunity to pass on a few of my own insights about workflow. I hope they're of use.

15.1 Use R Notebooks

OMG

I first started using R in the winter of 2015/2016. Right from the start, I learned how to code from within the [R Studio](#) environment. But within R Studio I was using simple scripts. No longer. I now use [R Notebooks](#) for just about everything, including my scientific projects, this [bookdown](#) project, and even my academic [webpage](#) and [blog](#). Nathan Stephens wrote a nice blog on [Why I love R Notebooks](#). I agree. This has fundamentally changed my workflow as a scientist. I only wish I'd learned about this before starting my dissertation project. So it goes...

Do yourself a favor, adopt R Notebooks into your workflow. Do it today. If you prefer to learn with videos, here's a nice intro by [Kristine Yu](#) and another one by [JJ Allaire](#). Try it out for like one afternoon and you'll be hooked.

15.2 Save your model fits

It's embarrassing how long it took for this to dawn on me.

Unlike classical statistics, Bayesian models using MCMC take a while to compute. Most of the simple models in McElreath's text take 30 seconds up to a couple minutes. If your data are small, well-behaved and of a simple structure, you might have a lot of wait times in that range in your future.

It hasn't been that way, for me.

Most of my data have a complicated multilevel structure and often aren't very well behaved. It's normal for my models to take an hour or several to fit. Once you start measuring your model fit times in hours, you do not want to fit these things more than once. So, it's not enough to document my code in a nice R Notebook file. I need to save my `brm()` fit objects in external files.

Consider this model. It's taken from Bürkner's vignette, [Estimating Multivariate Models with brms](#). It took about five minutes for my several-year-old laptop to fit.

```
library(brms)
data("BTdata", package = "MCMCglmm")
```

```
fit1 <-  
  brm(data = BTdata,  
    family = gaussian,  
    mvbind(tarsus, back) ~ sex + hatchdate + (1|p|fosternest) + (1|q|dam),  
    chains = 2, cores = 2,  
    seed = 15)
```

Five minutes isn't terribly long to wait, but still. I'd prefer to never have to wait for another five minutes, again. Sure, if I save my code in a document like this, I will always be able to fit the model again. But I can work smarter. Here I'll save my `fit1` object outside of R with the `save()` function.

```
save(fit1, file = "fit1.rda")
```

Hopefully y'all are savvy Bayesian R users and find this insultingly remedial. But if it's new to you like it was me, you can learn more about `.rda` files [here](#).

Now `fit1` is saved outside of R, I can safely remove it and then reload it.

```
rm(fit1)
```

```
load("fit1.rda")
```

The file took a fraction of a second to reload. Once reloaded, I can perform typical operations, like examine summaries of the model parameters or refreshing my memory on what data I used.

```
print(fit1)
```

```
## Family: MV(gaussian, gaussian)  
##   Links: mu = identity; sigma = identity  
##          mu = identity; sigma = identity  
## Formula: tarsus ~ sex + hatchdate + (1 | p | fosternest) + (1 | q | dam)  
##           back ~ sex + hatchdate + (1 | p | fosternest) + (1 | q | dam)  
## Data: BTdata (Number of observations: 828)  
## Samples: 2 chains, each with iter = 2000; warmup = 1000; thin = 1;  
##           total post-warmup samples = 2000  
  
##  
## Group-Level Effects:  
## ~dam (Number of levels: 106)  
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## sd(tarsus_Intercept)        0.48     0.05     0.39     0.58      1110 1.00  
## sd(back_Intercept)         0.24     0.07     0.10     0.38       461 1.00  
## cor(tarsus_Intercept,back_Intercept) -0.52     0.21    -0.92    -0.10      630 1.00  
##  
## ~fosternest (Number of levels: 104)  
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## sd(tarsus_Intercept)        0.27     0.05     0.16     0.38      838 1.00  
## sd(back_Intercept)         0.35     0.06     0.25     0.47       611 1.00  
## cor(tarsus_Intercept,back_Intercept) 0.68     0.20     0.20     0.98      234 1.01  
##  
## Population-Level Effects:  
##                               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## tarsus_Intercept      -0.41     0.07    -0.54    -0.28      1575 1.00  
## back_Intercept       -0.02     0.07    -0.15     0.11      2613 1.00  
## tarsus_sexMale        0.77     0.06     0.66     0.88      3229 1.00  
## tarsus_sexUNK         0.23     0.13    -0.03     0.49      3257 1.00  
## tarsus_hatchdate     -0.04     0.06    -0.16     0.07      1628 1.00  
## back_sexMale          0.01     0.07    -0.12     0.14      4479 1.00  
## back_sexUNK           0.15     0.15    -0.15     0.43      3548 1.00  
## back_hatchdate       -0.09     0.05    -0.20     0.01      2215 1.00
```

```
##  
## Family Specific Parameters:  
##  
## Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## sigma_tarsus 0.76 0.02 0.72 0.80 1966 1.00  
## sigma_back 0.90 0.02 0.86 0.95 2070 1.00  
##  
## Residual Correlations:  
##  
## Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat  
## rescor(tarsus,back) -0.05 0.04 -0.13 0.02 2649 1.00  
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

```
library(tidyverse)
```

```
fit1$data %>%  
  head()
```

```
##      tarsus sex hatchdate      dam fosternest      back  
## 1 -1.89229718 Fem -0.6874021 R187557 F2102 1.1464212  
## 2  1.13610981 Male -0.6874021 R187559 F1902 -0.7596521  
## 3  0.98468946 Male -0.4279814 R187568 A602  0.1449373  
## 4  0.37900806 Male -1.4656641 R187518 A1302 0.2555847  
## 5 -0.07525299 Fem -1.4656641 R187528 A2602 -0.3006992  
## 6 -1.13519543 Fem  0.3502805 R187945 C2302 1.5577219
```

As an alternative, Bürkner recently added a `file` argument in `brms:brm()` that will help you do this, too. The origins of the argument live [here](#). By default, `file` is set to `NULL`. To make use of the argument, specify a character string. `file` will then save your fitted model object in an external `.rds` file via the `saveRDS()` function. Let's give it a whirl, this time with an interaction.

```
fit2 <-  
  brm(data = BTdata,  
    family = gaussian,  
    mvbind(tarsus, back) ~ sex*hatchdate + (1|p|fosternest) + (1|q|dam),  
    chains = 2, cores = 2,  
    seed = 15,  
    file = "fit2")
```

Now `fit2` is saved outside of R, I can safely remove it and then reload it.

```
rm(fit2)
```

We might load `fit2` with the `readRDS()` function.

```
fit2 <- readRDS("fit2.rds")
```

Now we can work with `fit2` as desired.

```
fixef(fit2) %>%  
  round(digits = 3)  
  
##  
## Estimate Est.Error Q2.5 Q97.5  
## tarsus_Intercept      -0.409     0.070 -0.547 -0.268  
## back_Intercept        -0.011     0.067 -0.144  0.113  
## tarsus_sexMale         0.771     0.057  0.658  0.884
```

```
## tarsus_sexUNK          0.194   0.147 -0.104  0.485
## tarsus_hatchdate      -0.054   0.068 -0.186  0.077
## tarsus_sexMale:hatchdate 0.013   0.058 -0.101  0.125
## tarsus_sexUNK:hatchdate 0.058   0.122 -0.174  0.299
## back_sexMale           0.005   0.069 -0.128  0.138
## back_sexUNK             0.145   0.170 -0.190  0.472
## back_hatchdate         -0.052   0.061 -0.173  0.061
## back_sexMale:hatchdate -0.079   0.069 -0.209  0.053
## back_sexUNK:hatchdate  -0.033   0.143 -0.321  0.253
```

The `file` method has another handy feature. Let's remove `fit2` one more time to see.

```
rm(fit2)
```

If you've fit a `brm()` model once and saved the results with `file`, executing the same `brm()` code will not re-fit the model. Rather, it will just load and return the model from the `.rds` file.

```
fit2 <-
  brm(data = BTdata,
       family = gaussian,
       mvbind(tarsus, back) ~ sex*hatchdate + (1|p|fosternest) + (1|q|dam),
       chains = 2, cores = 2,
       seed = 15,
       file = "fit2")
```

It takes just a fraction of a second. Once again, we're ready to work with the fit.

```
fit2$formula
```

```
## tarsus ~ sex * hatchdate + (1 | p | fosternest) + (1 | q | dam)
## back ~ sex * hatchdate + (1 | p | fosternest) + (1 | q | dam)
```

And if you'd like to remind yourself what the name of that external file was, you can extract it from the `brm()` fit object.

```
fit2$file
```

```
## [1] "fit2.rds"
```

Also, see [Gavin Simpson's blog post](#) *A better way of saving and loading objects in R* for a discussion on the distinction between `.rda` and `.rds` files.

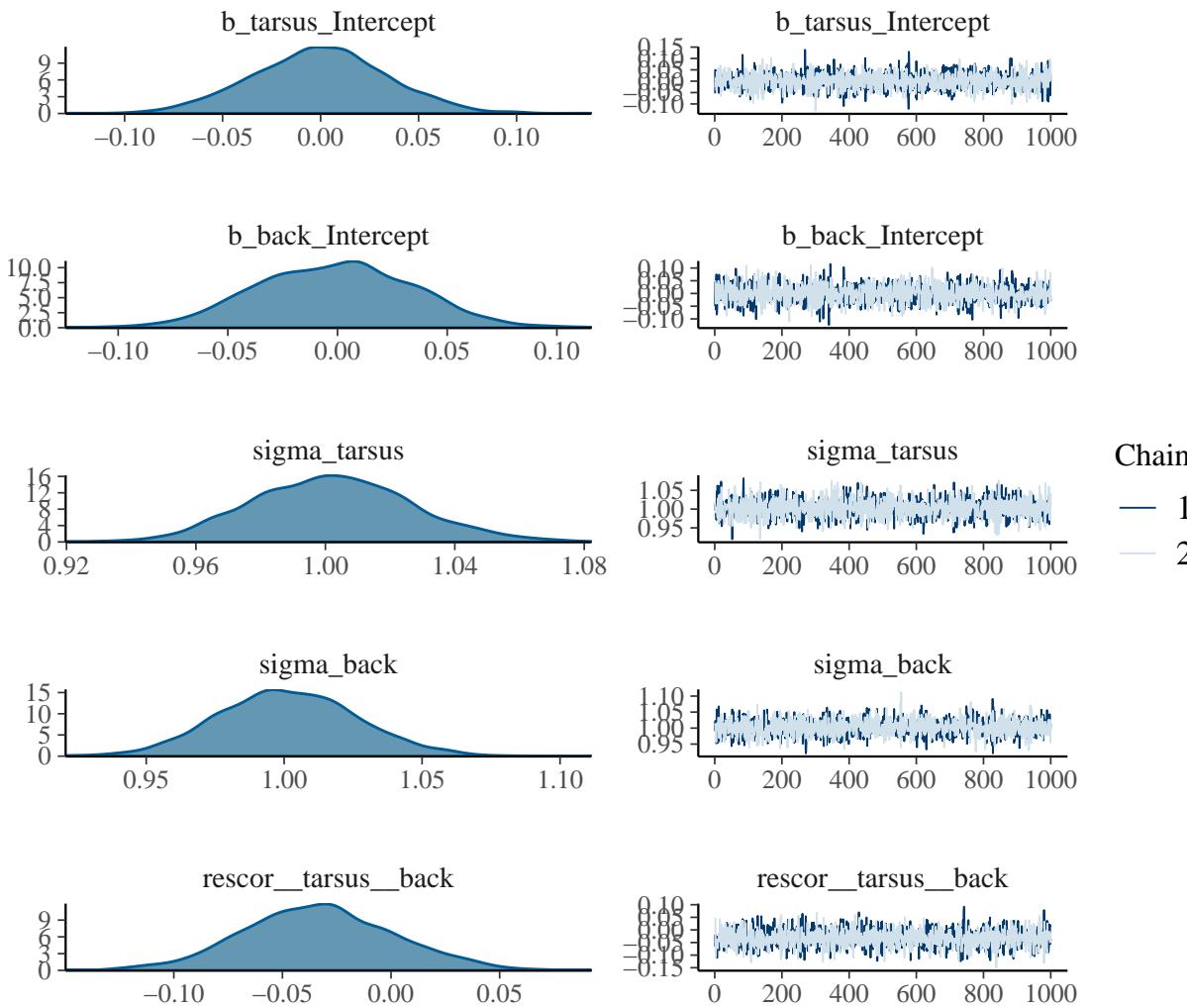
15.3 Build your models slowly

The model from Bürkner's vignette, `fit1`, was no joke. If you wanted to be verbose about it, it was a multilevel, multivariate, multivariable model. It had a cross-classified multilevel structure, two predictors (for each criterion), and two criteria. Not only is that a lot to keep track of, there's a whole lot of places for things to go wrong.

Even if that was the final model I was interested in as a scientist, I still wouldn't start with it. I'd build up incrementally, just to make sure nothing looked fishy. One place to start would be a simple intercepts-only model.

```
fit0 <-
  brm(mvbind(tarsus, back) ~ 1,
       data = BTdata, chains = 2, cores = 2,
       file = "fit0")
```

```
plot(fit0)
```



```
print(fit0)
```

```
## Family: MV(gaussian, gaussian)
##  Links: mu = identity; sigma = identity
##          mu = identity; sigma = identity
## Formula: tarsus ~ 1
##          back ~ 1
## Data: BTdata (Number of observations: 828)
## Samples: 2 chains, each with iter = 2000; warmup = 1000; thin = 1;
##          total post-warmup samples = 2000
##
## Population-Level Effects:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## tarsus_Intercept    -0.00      0.03    -0.07     0.07      2563 1.00
## back_Intercept     -0.00      0.04    -0.07     0.07      2242 1.00
##
## Family Specific Parameters:
##             Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## sigma_tarsus       1.00      0.02     0.96     1.05      2566 1.00
## sigma_back         1.00      0.03     0.95     1.05      2675 1.00
##
## Residual Correlations:
##                         Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
## rescor(tarsus,back)   -0.03      0.03    -0.10     0.04      2453 1.00
```

```
##  
## Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample  
## is a crude measure of effective sample size, and Rhat is the potential  
## scale reduction factor on split chains (at convergence, Rhat = 1).
```

If the chains look good and the summary statistics look like what I'd expect, I'm on good footing to keep building up to the model I really care about. The results from this model, for example, suggest that both criteria were standardized (i.e., intercepts at 0 and σ s at 1). If that wasn't what I intended, I'd rather catch it here than spend five minutes fitting the more complicated `fit1` model, the parameters for which are sufficiently complicated that I may have had trouble telling what scale the data were on.

Note, this is not the same as [p-hacking](#) or [wandering aimlessly down the garden of forking paths](#). We are not chasing the flashiest model to put in a paper. Rather, this is just [good pragmatic data science](#). If you start off with a theoretically-justified but complicated model and run into computation problems or produce odd-looking estimates, it won't be clear where things went awry. When you build up, step by step, it's easier to catch data cleaning failures, coding goofs and the like.

So, when I'm working on a project, I fit one or a few simplified models before fitting my complicated model of theoretical interest. This is especially the case when I'm working with model types that are new to me or that I haven't worked with in a while. I document each step in my R Notebook files and I save the fit objects for each in external files. I have caught surprises this way. Hopefully this will help you catch your mistakes, too.

15.4 Look at your data

Relatedly, and perhaps even a precursor, you should [always plot your data](#) before fitting a model. There were plenty examples of this in the text, but it's worth of making explicit. Simple summary statistics are great, but they're not enough. For an entetrtaining exposition, check out [Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing](#). Though it might make for a great cocktail party story, I'd hate to pollute the literature with a linear model based on a set of dinosaur-shaped data.

15.5 Use the `0 + intercept` syntax

We covered this a little in the last couple chapters, but it's easy to miss. If your real-world model has predictors (i.e., isn't an intercept-only model), it's important to keep track of how you have centered those predictors. When you specify a prior for a brms `Intercept` (i.e., an intercept resulting from the `y ~ x` or `y ~ 1 + x` style of syntax), that prior is applied under the presumption all the predictors are mean centered. In the *Population-level ('fixed')* effects subsection of the `set_prior` section of the [brms reference manual](#) (version 2.8.0), we read:

Note that technically, this prior is set on an intercept that results when internally centering all population-level predictors around zero to improve sampling efficiency. On this centered intercept, specifying a prior is actually much easier and intuitive than on the original intercept, since the former represents the expected response value when all predictors are at their means. To treat the intercept as an ordinary population-level effect and avoid the centering parameterization, use `0 + intercept` on the right-hand side of the model formula. (p. 153)

We get a little more information from the *Parameterization of the population-level intercept* subsection of the `brmsformula` section:

This behavior can be avoided by using the reserved (and internally generated) variable `intercept`. Instead of `y ~ x`, you may write `y ~ 0 + intercept + x`. This way, priors can be defined on the real intercept, directly. In addition, the intercept is just treated as an ordinary population-level effect and thus priors defined on `b` will also apply to it. Note that this parameterization may be less efficient than the default parameterization discussed above. (p. 32)

We didn't bother with this for most of the project because our priors on the `Intercept` were often vague and the predictors were often on small enough scales (e.g., the mean of a dummy variable is close to 0) that it just didn't matter. But this will not always be the case. Set your `Intercept` priors with care.

There's also the flip side of the issue. If there's no strong reason not to, consider mean-centering or even standardizing your predictors. Not only will that solve the `Intercept` prior issue, but it often results in more meaningful parameter estimates.

15.6 Annotate your workflow

In a typical model-fitting file, I'll load my data, perhaps transform the data a bit, fit several models, and examine the output of each with trace plots, model summaries, information criteria, and the like. In my early days, I just figured each of these steps were self-explanatory.

Nope.

["In every project you have at least one other collaborator; future-you. You don't want future-you to curse past-you."](#)

My experience was that even a couple weeks between taking a break from a project and restarting it was enough time to make my earlier files confusing. **And they were my files.** I now start each R Notebook document with an introductory paragraph or two explaining exactly what the purpose of the file is. I separate my major sections by [headers and subheaders](#). My working R Notebook files are peppered with bullets, sentences, and full on paragraphs between code blocks.

15.7 Annotate your code

This idea is implicit in McElreath's text. But it's easy to miss the message. I know I did, at first. I find this is especially important for data wrangling. I'm a tidyverse guy and, for me, the big-money verbs like `mutate()`, `gather()`, `select()`, `filter()`, `group_by()`, and `summarise()` take care of the bulk of my data wrangling. But every once and a while I need to do something less common, like with `str_extract()` or `case_when()`. And when I end up using a new or less familiar function, I typically annotate right in the code and even sometimes leave a hyperlink to some [R-bloggers](#) post or [stackoverflow](#) question that explained how to use it.

15.8 Break up your workflow

I've also learned to break up my projects into multiple R Notebook files. If you have a small project for which you just want a quick and dirty plot, fine, do it all in one file. My typical project has:

- A primary data cleaning file
- A file with basic descriptive statistics and the like
- At least one primary analysis file
- Possible secondary and tertiary analysis files
- A file or two for my major figures
- A file explaining and depicting my priors, often accompanied by my posteriors, for comparison

Putting all that information in one R Notebook file would be overwhelming. Your workflow might well look different, but hopefully you get the idea. You don't want working files with thousands of lines of code.

And mainly to keep Jenny Bryan from [setting my computer on fire](#), I'm also getting into the habit of organizing all these interconnected files with help from [R Studio Projects](#), which you can learn even more about from [this chapter](#) in *R4DS*.

15.9 Read Gelman's blog

Yes, [that Gelman](#).

Actually, I started reading [Gelman's blog](#) around the same time I dove into [McElreath's text](#). But if this isn't the case for you, it's time to correct that evil. My graduate mentor often recalled how transformative his first academic conference was. He was an undergrad at the time and it was his first experience meeting and talking with the people whose names he'd seen in his text books. He learned that science was an ongoing conversation among living scientists and—at that time—the best place to take part in that conversation was at conferences. Times keep changing. Nowadays, the living conversation of science occurs online on social media and in blogs. One of the hottest places to find scientists conversing about Bayesian statistics and related methods is [Gelman's blog](#). The posts are great. But a lot of the action is in the comments sections, too.

15.10 Check out other social media, too

If you're not on it, consider joining academic [twitter](#). The word on the street is correct. Twitter can be rage-fueled [dumpster fire](#). But if you're selective about who you follow, it's a great place to lean from and connect with your academic heroes. If you're a fan of this project, here's a list of some of the people you might want to follow:

- [Richard McElreath](#)
- [Paul Bürkner](#)
- [Aki Vehtari](#)
- [Dan Simpson](#)
- [Michael Bentacourt](#)
- [Hadley Wickham](#)
- [Yihui Xie](#)
- [Jenny Bryan](#)
- [Roger Peng](#)
- [Mara Averick](#)
- [Matthew Kay](#)
- [Matti Vuorre](#)
- [Tristan Mahr](#)
- [Danielle Navarro](#)

[I'm on twitter](#), too.

If you're on facebook and in the social sciences, you might check out the [Bayesian Inference in Psychology](#) group. It hasn't been terribly active, as of late. But there are a lot of great folks to connect with, there.

I've already mentioned Gelman's blog. [McElreath has one, too](#). He posts infrequently, but it's usually pretty good when he does.

Also, do check out the [Stan Forums](#). They have a special `brms` tag there, under which you can find all kinds of hot `brms` talk.

But if you're new to the world of asking for help with your code online, you might acquaint yourself with the notion of a [minimally reproducible example](#). In short, a good minimally reproducible example helps others help you. If you fail to do this, prepare for some skark.

15.11 Parting wisdom

Okay, that's enough from me. Let's start wrapping this project up with some McElreath.

There is an aspect of science that you do personally control: openness. [Pre-plan your research together with the statistical analysis](#). Doing so will improve both the research design and the statistics. Document it in the form of a mock analysis that you would not be ashamed to share with a colleague. Register it publicly, perhaps in a simple repository, like [Github](#) or any other. But [your webpage](#) will do just fine, as well. Then collect the data. Then analyze the data as planned. If you must change the plan, that's fine. But document the changes and justify them. [Provide all of the data and scripts necessary to repeat your analysis](#). Do not provide scripts and data "on request," but rather put them online so reviewers of your paper can access them without your interaction. There are of course cases in which full data cannot be released, due to privacy concerns. But the bulk of science is not of that sort.

The data and its analysis are the scientific product. The paper is just an advertisement. If you do your honest best to design, conduct, and document your research, so that others can build directly upon it, you can make a difference. (p. 443)

Toward that end, also check out the [OSF](#) and their YouTube channel, [here](#). Katie Corker gets the last words: "[Open science is stronger because we're doing this together](#)."

Reference

McElreath, R. (2016). *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman & Hall/CRC Press.

Session info

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS High Sierra 10.13.6
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats      graphics   grDevices  utils       datasets   methods    base
##
## other attached packages:
## [1]forcats_0.3.0  stringr_1.4.0  dplyr_0.8.0.1  purrr_0.2.5   readr_1.1.1    tidyverse_0.8.1
## [7]tibble_2.1.1   ggplot2_3.1.1  tidyverse_1.2.1 brms_2.8.8     Rcpp_1.0.1
##
## loaded via a namespace (and not attached):
## [1] nlme_3.1-137      matrixStats_0.54.0   xts_0.10-2        lubridate_1.7.4
## [5] threejs_0.3.1     httr_1.3.1          rprojroot_1.3-2   rstan_2.18.2
## [9] tools_3.5.1       backports_1.1.4     R6_2.3.0          DT_0.4
## [13] lazyeval_0.2.2    colorspace_1.3-2    withr_2.1.2       tidyselect_0.2.5
## [17] gridExtra_2.3     prettyunits_1.0.2   processx_3.2.1    Brobdingnag_1.2-6
## [21] compiler_3.5.1    rvest_0.3.2         cli_1.0.1         xml2_1.2.0
## [25] shinyjs_1.0        labeling_0.3        colourpicker_1.0  bookdown_0.9
## [29] scales_1.0.0      dygraphs_1.1.1.5   mvtnorm_1.0-10    ggridges_0.5.0
## [33] callr_3.1.0       digest_0.6.18       StanHeaders_2.18.0-1 rmarkdown_1.10
## [37] base64enc_0.1-3   pkgconfig_2.0.2     htmltools_0.3.6   htmlwidgets_1.2
## [41] rlang_0.3.4       readxl_1.1.0       rstudioapi_0.7    shiny_1.1.0
## [45] generics_0.0.2    zoo_1.8-2          jsonlite_1.5      crosstalk_1.0.0
## [49] gtools_3.8.1      inline_0.3.15      magrittr_1.5      loo_2.1.0
## [53] bayesplot_1.6.0   Matrix_1.2-14      munsell_0.5.0     abind_1.4-5
## [57] stringi_1.4.3    yaml_2.1.19       pkgbuild_1.0.2    plyr_1.8.4
## [61] grid_3.5.1        parallel_3.5.1    promises_1.0.1    crayon_1.3.4
## [65] miniUI_0.1.1.1   lattice_0.20-35   haven_1.1.2      hms_0.4.2
## [69] knitr_1.20        ps_1.2.1          pillar_1.3.1     igraph_1.2.1
## [73] markdown_0.8       shinystan_2.5.0   reshape2_1.4.3    stats4_3.5.1
## [77] rstantools_1.5.1  glue_1.3.1.9000  evaluate_0.10.1   modelr_0.1.2
## [81] httpuv_1.4.4.2   cellranger_1.1.0  gtable_0.3.0      assertthat_0.2.0
## [85] xfun_0.3          mime_0.5          xtable_1.8-2     broom_0.5.1
## [89] coda_0.19-2      later_0.7.3       rsconnect_0.8.8   shinythemes_1.1.1
## [93] bridgesampling_0.6-0
```