13 Models With Memory

In the year 1985, Clive Wearing lost his mind, but not his music.[185] Wearing was a musicologist and accomplished musician, but the same virus that causes cold sores, *Herpes simplex*, snuck into his brain and ate his hippocampus. The result was chronic anterograde amnesia—he cannot form new long-term memories. He remembers how to play the piano, though he cannot remember that he played it 5 minutes ago. Wearing now lives moment to moment, unaware of anything more than a few minutes into the past. Every cup of coffee is the first he has ever had.

Many statistical models also have anterograde amnesia. As the models move from one cluster—individual, group, location—in the data to another, estimating parameters for each cluster, they forget everything about the previous clusters. They behave this way, because the assumptions force them to. Any of the models from previous chapters that used dummy variables (page 157) to handle categories are programmed for amnesia. These models implicitly assume that nothing learned about any one category informs estimates for the other categories—the parameters are independent of one another and learn from completely separate portions of the data. This would be like forgetting you had ever been in a café, each time you go to a new café. Cafés do differ, but they are also alike.

Anterograde amnesia is bad for learning about the world. We want models that instead use all of the information in savvy ways. This does not mean treating all clusters as if they were the same. Instead it means learning simultaneously about each cluster while learning about the population of clusters. Doing both estimation tasks at the same time allows us to transfer information across clusters, and that transfer improves accuracy. That is the value of remembering.

Consider cafés again. Suppose we program a robot to visit two cafés, order coffee, and estimate the waiting times at each. The robot begins with a vague prior for the waiting times, say with a mean of 5 minutes and a standard deviation of 1. After ordering a cup of coffee at the first café, the robot observes a waiting time of 4 minutes. It updates its prior, using Bayes' theorem of course, with this information. This gives it a posterior distribution for the waiting time at the first café.

Now the robot moves on to a second café. When this robot arrives at the next café, what is its prior? It could just use the posterior distribution from the first café as its prior for the second café. But that implicitly assumes that the two cafés have the same average waiting time. Cafés are all pretty much the same, but they aren't identical. Likewise, it doesn't make much sense to ignore the observation from the first café. That would be anterograde amnesia.

So how can the coffee robot do better? It needs to represent the population of cafés and learn about that population. The distribution of waiting times in the population becomes the prior for each café. But unlike priors in previous chapters, this prior is actually learned

from the data. This means the robot tracks a parameter for each café as well as at least two parameters to describe the population of cafés: an average and a standard deviation. As the robot observes waiting times, it updates everything: the estimates for each café as well as the estimates for the population. If the population seems highly variable, then the prior is flat and uninformative and, as a consequence, the observations at any one café do very little to the estimate at another. If instead the population seems to contain little variation, then the prior is narrow and highly informative. An observation at any one café will have a big impact on estimates at any other café.

In this chapter, you'll see the formal version of this argument and how it leads us to **MULTILEVEL MODELS**. These 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. I mentioned them in Chapter 1. They are worth repeating.

(1) *Improved estimates for repeat sampling.* When more than one observation arises from the same individual, location, or time, then traditional, single-level models either maximally underfit or overfit the data.

(2) *Improved estimates for imbalance in sampling.* When some individuals, locations, or times are sampled more than others, multilevel models automatically cope with differing uncertainty across these clusters. This prevents over-sampled clusters from unfairly dominating inference.

(3) *Estimates of variation.* If our research questions include variation among individuals or other groups within the data, then multilevel models are a big help, because they model variation explicitly.

(4) *Avoid averaging, retain variation.* Frequently, scholars pre-average some data to construct variables. This can be dangerous, because averaging removes variation, and there are also typically several different ways to perform the averaging. Averaging therefore both manufactures false confidence and introduces arbitrary data transformations. Multilevel models allow us to preserve the uncertainty and avoid data transformations.

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 strategy, it becomes much easier to incorporate related tricks such as allowing for measurement error in the data and even modeling missing data itself (Chapter 15).

There are costs of the multilevel approach. The first is that we have to make some new assumptions. We have to define the distributions from which the characteristics of the clusters arise. Luckily, conservative maximum entropy distributions do an excellent job in this context. Second, there are new estimation challenges that come with the full multilevel approach. These challenges lead us headfirst into MCMC estimation. Third, multilevel models can be hard to understand, because they make predictions at different levels of the data. In

many cases, we are interested in only one or a few of those levels, and as a consequence, model comparison using metrics like DIC and WAIC becomes more subtle. The basic logic remains unchanged, but now we have to make more decisions about which parameters in the model we wish to focus on.

This chapter has the following progression. First, we'll work through an extended example of building and fitting a multilevel model for clustered data. Then we'll simulate clustered data, to demonstrate the improved accuracy the approach delivers. This improved accuracy arises from the same underfitting and overfitting trade-off you met in Chapter 7. Then we'll finish by looking at contexts in which there is more than one type of clustering. All of this work lays a foundation for more advanced multilevel examples in the next two chapters.

Rethinking: A model by any other name. Multilevel models go by many different names, and some statisticians use the same names for different specialized variants, while others use them all interchangeably. The most common synonyms for "multilevel" are HIERARCHICAL and MIXED EFFECTS. The type of parameters that appear in multilevel models are most commonly known as RANDOM EFFECTS, which itself can mean very different things to different analysts and in different contexts.[186] And even the innocent term "level" can mean different things to different people. There's really no cure for this swamp of vocabulary aside from demanding a mathematical or algorithmic definition of the model. Otherwise, there will always be ambiguity.

13.1. Example: Multilevel tadpoles

The heartwarming focus of this example are experiments exploring Reed frog (*Hyperolius spinigularis*) tadpole mortality.[187] The natural history background to these data is very interesting. Take a look at the full paper, if amphibian life history dynamics interests you. But even if it doesn't, load the data and acquaint yourself with the variables:

R code 13.1

```
library(rethinking)
data(reedfrogs)
d <- reedfrogs
str(d)
```

```
'data.frame': 48 obs. of  5 variables:
 $ density : int  10 10 10 10 10 10 10 10 10 10 ...
 $ pred    : Factor w/ 2 levels "no","pred": 1 1 1 1 1 1 1 2 2 ...
 $ size    : Factor w/ 2 levels "big","small": 1 1 1 1 2 2 2 2 1 1 ...
 $ surv    : int  9 10 7 10 9 9 10 9 4 9 ...
 $ propsurv: num  0.9 1 0.7 1 0.9 0.9 1 0.9 0.4 0.9 ...
```

For now, we'll only be interested in number surviving, `surv`, out of an initial count, `density`. In the practice at the end of the chapter, you'll consider the other variables, which are experimental manipulations.

There is a lot of variation in these data. Some of the variation comes from experimental treatment. But a lot of it comes from other sources. Think of each row as a "tank," an experimental environment that contains tadpoles. There are lots of things peculiar to each tank that go unmeasured, and these unmeasured factors create variation in survival across tanks, even when all the predictor variables have the same value. These tanks are an example of a *cluster* variable. Multiple observations, the tadpoles in this case, are made within each cluster.

So we have repeat measures and heterogeneity across clusters. If we ignore the clusters, assigning the same intercept to each of them, then we risk ignoring important variation in baseline survival. This variation could mask association with other variables. If we instead estimate a unique intercept for each cluster, using a dummy variable for each tank, we instead practice anterograde amnesia. After all, tanks are different but each tank does help us estimate survival in the other tanks. So it doesn't make sense to forget entirely, moving from one tank to another.

A multilevel model, in which we simultaneously estimate both an intercept for each tank and the variation among tanks, is what we want. This will be a **VARYING INTERCEPTS** model. Varying intercepts are the simplest kind of **VARYING EFFECTS**.[188] For each cluster in the data, we use a unique intercept parameter. This is no different than the categorical variable examples from previous chapters, except now we also adaptively learn the prior that is common to all of these intercepts. This adaptive learning is the absence of amnesia discussed at the start of the chapter. When what we learn about each cluster informs all the other clusters, we learn the prior simultaneous to learning the intercepts.

Here is a model for predicting tadpole mortality in each tank, using the regularizing priors of earlier chapters:

$$\begin{aligned} S_i &\sim \text{Binomial}(N_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{TANK}[i]} \qquad \text{[unique log-odds for each tank]} \\ \alpha_j &\sim \text{Normal}(0, 1.5) \quad \text{, for } j = 1..48 \end{aligned}$$

And you can approximate this posterior using `ulam` as in previous chapters:

R code 13.2

```
# make the tank cluster variable
d$tank <- 1:nrow(d)

dat <- list(
    S = d$surv,
    N = d$density,
    tank = d$tank )

# approximate posterior
m13.1 <- ulam(
    alist(
        S ~ dbinom( N , p ) ,
        logit(p) <- a[tank] ,
        a[tank] ~ dnorm( 0 , 1.5 )
    ), data=dat , chains=4 , log_lik=TRUE )
```

If you inspect the posterior, `precis(m13.1,depth=2)`, you'll see 48 different intercepts, one for each tank. To get each tank's expected survival probability, just take one of the `a` values and then use the logistic transform. So far there is nothing new here.

Now let's do the multilevel model, which adaptively pools information across tanks. All that is required to enable adaptive pooling is to make the prior for the `a` parameters a function of some new parameters. Here is the multilevel model, in mathematical form, with the

changes from the previous model highlighted in blue:

$$
\begin{aligned}
S_i &\sim \text{Binomial}(N_i, p_i) \\
\text{logit}(p_i) &= \alpha_{\text{TANK}[i]} \\
\alpha_j &\sim \text{Normal}(\bar{\alpha}, \sigma) && \text{[adaptive prior]} \\
\bar{\alpha} &\sim \text{Normal}(0, 1.5) && \text{[prior for average tank]} \\
\sigma &\sim \text{Exponential}(1) && \text{[prior for standard deviation of tanks]}
\end{aligned}
$$

Notice that the prior for the tank intercepts is now a function of two parameters, $\bar{\alpha}$ and σ. You can say $\bar{\alpha}$ like "bar alpha." The bar means average. These two parameters inside the prior is where the "multi" in multilevel arises.[189] The Gaussian distribution with mean $\bar{\alpha}$ and standard deviation σ is the prior for each tank's intercept. But that prior itself has priors for $\bar{\alpha}$ and σ. So there are two *levels* in the model, each resembling a simpler model. In the top level, the outcome is S, the parameters are the vector α, and the prior is $\alpha_j \sim \text{Normal}(\bar{\alpha}, \sigma)$. In the second level, the "outcome" variable is the vector of intercept parameters, α. The parameters are $\bar{\alpha}$ and σ, and their priors are $\bar{\alpha} \sim \text{Normal}(0, 1.5)$ and $\sigma \sim \text{Exponential}(1)$.

These two parameters, $\bar{\alpha}$ and σ, are often referred to as **HYPERPARAMETERS**. They are parameters for parameters. And their priors are often called **HYPERPRIORS**. In principle, there is no limit to how many "hyper" levels you can install in a model. For example, different populations of tanks could be embedded within different regions of habitat. But in practice there are limits, both because of computation and our ability to understand the model.

Rethinking: Why Gaussian tanks? In the multilevel tadpole model, the population of tanks is assumed to be Gaussian. Why? The least satisfying answer is "convention." The Gaussian assumption is extremely common. A more satisfying answer is "pragmatism." The Gaussian assumption is easy to work with, and it generalizes easily to more than one dimension. This generalization will be important for handling varying slopes in the next chapter. But my preferred answer is instead "entropy." If all we are willing to say about a distribution is the mean and variance, then the Gaussian is the most conservative assumption (Chapter 10). Using a Gaussian here does not force the resulting posterior distribution of α parameters to be symmetric or have a Gaussian shape. The only information in a Gaussian prior (or likelihood) is finite variance. The distribution looks symmetric, because if you don't say how it is skewed, then symmetric is the maximum entropy shape. Above all, there is no rule requiring the Gaussian distribution of varying effects. So if you have a good reason to use another distribution, then do so. The practice problems at the end of the chapter provide an example.

Computing the posterior computes both levels simultaneously, in the same way that our robot at the start of the chapter learned both about each café and the variation among cafés. But you cannot fit this model with `quap`. Why? Because the probability of the data must now average over the level 2 parameters $\bar{\alpha}$ and σ. But `quap` just hill climbs, using static values for all of the parameters. It can't see the levels. For more explanation, see the Overthinking box further down. You can however fit this model with `ulam`:

R code 13.3

```
m13.2 <- ulam(
    alist(
        S ~ dbinom( N , p ) ,
        logit(p) <- a[tank] ,
        a[tank] ~ dnorm( a_bar , sigma ) ,
        a_bar ~ dnorm( 0 , 1.5 ) ,
```

```
        sigma ~ dexp( 1 )
    ), data=dat , chains=4 , log_lik=TRUE )
```

This model provides posterior distributions for 50 parameters: one overall sample intercept $\bar{\alpha}$, the standard deviation among tanks σ, and then 48 per-tank intercepts. Let's check WAIC though to see the effective number of parameters. We'll compare the earlier model, `m13.1`, with the new multilevel model:

R code 13.4

```
compare( m13.1 , m13.2 )
```

```
       WAIC   SE dWAIC  dSE pWAIC weight
m13.2 200.0 7.19   0.0   NA  20.9      1
m13.1 215.9 4.43  15.9 4.03  26.2      0
```

There are two facts to note here. First, the multilevel model has only 21 effective parameters. There are 28 fewer effective parameters than actual parameters, because the prior assigned to each intercept shrinks them all towards the mean $\bar{\alpha}$. In this case, the prior is reasonably strong. Check the mean of `sigma` with `precis` and you'll see it's around 1.6. This is a **REGULARIZING PRIOR**, like you've used in previous chapters, but now the amount of regularization has been learned from the data itself.[190] Second, notice that the multilevel model `m13.2` has fewer effective parameters than the ordinary fixed model `m13.1`. This is despite the fact that the ordinary model has fewer actual parameters, only 48 instead of 50. The extra two parameters in the multilevel model allowed it to learn a more aggressive regularizing prior, to adaptively regularize. This resulted in a less flexible posterior and therefore fewer effective parameters.

Overthinking: QUAP fails, MCMC succeeds. Why doesn't simple quadratic approximation, using for example `quap`, work with multilevel models? When a prior is itself a function of parameters, there are two levels of uncertainty. This means that the probability of the data, conditional on the parameters, must average over each level. Ordinary quadratic approximation cannot handle the averaging in the likelihood, because in general it's not possible to derive an analytical solution. That means there is no unified function for calculating the log-posterior. So your computer cannot directly find its minimum (the maximum of the posterior).

Some other computational approach is needed. It is possible to extend the mode-finding optimization strategy to these models, but we don't want to be stuck with optimization in general. One reason is that the posterior of these models is routinely non-Gaussian. Another is that optimization approaches tend to be more fragile than MCMC. Stan actually has optimization routines. See `?optimizing`.

To appreciate the impact of this adaptive regularization, let's plot and compare the posterior means from models `m13.1` and `m13.2`. The code that follows is long, only because it decorates the plot with informative labels. The basic code is just the first part, which extracts samples and computes means.

R code 13.5

```
# extract Stan samples
post <- extract.samples(m13.2)

# compute median intercept for each tank
```

```
# also transform to probability with logistic
d$propsurv.est <- logistic( apply( post$a , 2 , mean ) )

# display raw proportions surviving in each tank
plot( d$propsurv , ylim=c(0,1) , pch=16 , xaxt="n" ,
    xlab="tank" , ylab="proportion survival" , col=rangi2 )
axis( 1 , at=c(1,16,32,48) , labels=c(1,16,32,48) )

# overlay posterior means
points( d$propsurv.est )

# mark posterior mean probability across tanks
abline( h=mean(inv_logit(post$a_bar)) , lty=2 )

# draw vertical dividers between tank densities
abline( v=16.5 , lwd=0.5 )
abline( v=32.5 , lwd=0.5 )
text( 8 , 0 , "small tanks" )
text( 16+8 , 0 , "medium tanks" )
text( 32+8 , 0 , "large tanks" )
```

You can see the result in FIGURE 13.1. The horizontal axis is tank index, from 1 to 48. The vertical is proportion of survivors in a tank. The filled blue points show the raw proportions, computed from the observed counts. These values are already present in the data frame, in the `propsurv` column. The black circles are instead the varying intercept medians. The horizontal dashed line at about 0.8 is the estimated median survival proportion in the population of tanks, α. It is not the same as the empirical mean survival. The vertical gray lines divide tanks with different initial counts of tadpoles—10 (left), 25 (middle), and 35 (right).

First, notice that in every case, the multilevel estimate is closer to the dashed line than the raw empirical estimate is. It's as if the entire distribution of black circles has been shrunk towards the dashed line at the center of the data, leaving the blue points behind on the outside. This phenomenon is sometimes called **SHRINKAGE**, and it results from regularization (as in Chapter 7). Second, notice that the estimates for the smaller tanks have shrunk farther from the blue points. As you move from left to right in the figure, the initial densities of tadpoles increase from 10 to 25 to 35, as indicated by the vertical dividers. In the smallest tanks, it is easy to see differences between the open estimates and empirical blue points. But in the largest tanks, there is little difference between the blue points and open circles. Varying intercepts for the smaller tanks, with smaller sample sizes, shrink more. Third, note that the farther a blue point is from the dashed line, the greater the distance between it and the corresponding multilevel estimate. Shrinkage is stronger, the further a tank's empirical proportion is from the global average α.

All three of these phenomena arise from a common cause: pooling information across clusters (tanks) to improve estimates. What **POOLING** means here is that each tank provides information that can be used to improve the estimates for all of the other tanks. Each tank helps in this way, because we made an assumption about how the varying log-odds in each tank related to all of the others. We assumed a distribution, the normal distribution in this

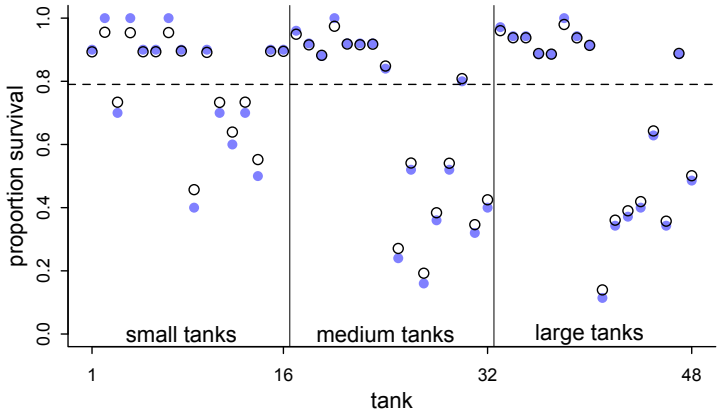

FIGURE 13.1. Empirical proportions of survivors in each tadpole tank, shown by the filled blue points, plotted with the 48 per-tank parameters from the multilevel model, shown by the black circles. The dashed line locates the average proportion of survivors across all tanks. The vertical lines divide tanks with different initial densities of tadpoles: small tanks (10 tadpoles), medium tanks (25), and large tanks (35). In every tank, the posterior mean from the multilevel model is closer to the dashed line than the empirical proportion is. This reflects the pooling of information across tanks, to help with inference about each tank.

case. Once we have a distributional assumption, we can use Bayes' theorem to optimally (in the small world only) share information among the clusters.

What does the inferred population distribution of survival look like? We can visualize it by sampling from the posterior distribution, as usual. First we'll plot 100 Gaussian distributions, one for each of the first 100 samples from the posterior distribution of both α and σ. Then we'll sample 8000 new log-odds of survival for individual tanks. The result will be a posterior distribution of variation in survival in the population of tanks. Before we do the sampling though, remember that "sampling" from a posterior distribution is not a simulation of empirical sampling. It's just a convenient way to characterize and work with the uncertainty in the distribution. Now the sampling:

R code 13.6

```
# show first 100 populations in the posterior
plot( NULL , xlim=c(-3,4) , ylim=c(0,0.35) ,
    xlab="log-odds survive" , ylab="Density" )
for ( i in 1:100 )
    curve( dnorm(x,post$a_bar[i],post$sigma[i]) , add=TRUE ,
    col=col.alpha("black",0.2) )

# sample 8000 imaginary tanks from the posterior distribution
sim_tanks <- rnorm( 8000 , post$a_bar , post$sigma )
```

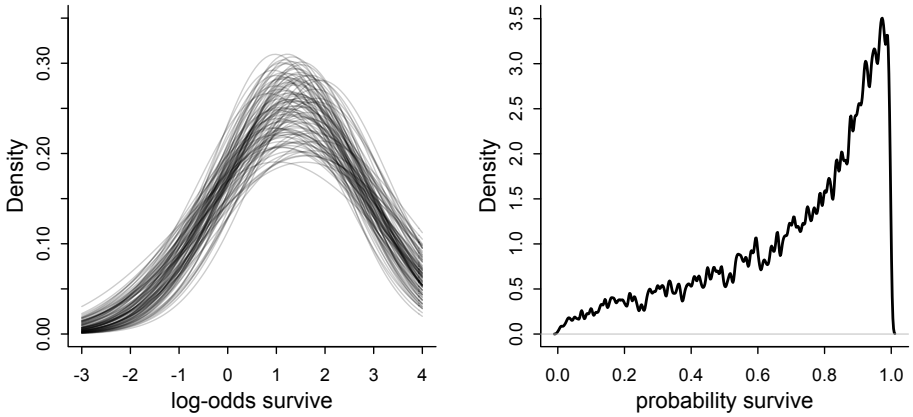

FIGURE 13.2. The inferred population of survival across tanks. Left: 100 Gaussian distributions of the log-odds of survival, sampled from the posterior of `m13.2`. Right: Survival probabilities for 8000 new simulated tanks, averaging over the posterior distribution on the left.

```
# transform to probability and visualize
dens( inv_logit(sim_tanks) , lwd=2 , adj=0.1 )
```

The results are displayed in FIGURE 13.2. Notice that there is uncertainty about both the location, α, and scale, σ, of the population distribution of log-odds of survival. All of this uncertainty is propagated into the simulated probabilities of survival.

Rethinking: Varying intercepts as over-dispersion. In the previous chapter (page 381), the beta-binomial and gamma-Poisson models were presented as ways for coping with **OVER-DISPERSION** of count data. Varying intercepts accomplish the same thing, allowing count outcomes to be over-dispersed. They accomplish this, because when each observed count gets its own unique intercept, but these intercepts are pooled through a common distribution, the predictions expect over-dispersion just like a beta-binomial or gamma-Poisson model would. Compared to a beta-binomial or gamma-Poisson model, a binomial or Poisson model with a varying intercept on every observed outcome will often be easier to estimate and easier to extend. There will be an example of this approach, later in this chapter.

Overthinking: Priors for variance components. The examples in this book use weakly regularizing exponential priors for variance components, the σ parameters that estimate the variation across clusters in the data. These exponential priors work very well in routine multilevel modeling. They express only a rough notion of an average standard deviation and regularize towards zero. But there are two common contexts in which they can be problematic. First, sometimes there isn't much information in the data with which to estimate the variance. For example, if you only have 5 clusters, then that's something like trying to estimate a variance with 5 data points. In that case, you might need something much more informative. Second, in non-linear models with logit and log links, floor and ceiling effects sometimes render extreme values of the variance equally plausible as more realistic values. In

such cases, the trace plot for the variance parameters may swing around over very large values. It can do this, because the exponential prior has a long tail. Such large values are typically *a priori* impossible. Often, the chain will still sample validly, but it might be highly inefficient, exhibiting small `n_eff` values and possibly many divergent transitions.

To improve such a model, instead of using exponential priors for the variance components, you can use half-Normal priors or some other prior with a thin tail. A half-Normal is a Normal distribution with all mass above zero. It is just cut off below zero. For example:

$$\begin{aligned} S_i &\sim \text{Binomial}(N_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{TANK}[i]} \\ \alpha_j &\sim \text{Normal}(\bar{\alpha}, \sigma) \\ \alpha &\sim \text{Normal}(0, 1.5) \\ \sigma &\sim \text{Half-Normal}(0, 1) \end{aligned}$$

Inside an `ulam` formula, you'd use `dhalfnorm`. Inside a Stan model, you just assign a lower bound to the parameter of `lower=0`.

13.2. 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, so if it still seems mysterious, this section aims to further relate the properties of multilevel estimates to the foundational underfitting/overfitting dilemma from Chapter 7.

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.[191] On average, the varying effects actually provide a better estimate of the individual tank (cluster) means. The reason that the varying intercepts provide better estimates is that they do a better job of trading off underfitting and overfitting.

To understand this in the context of the reed frog example, suppose that instead of experimental tanks we had natural ponds, so that we might be concerned with making predictions for the same clusters in the future. We'll approach the problem of predicting future survival in these ponds, from three perspectives:

(1) Complete pooling. This means we assume that the population of ponds is invariant, the same as estimating a common intercept for all ponds.
(2) No pooling. This means we assume that each pond tells us nothing about any other pond. This is the model with amnesia.
(3) Partial pooling. This means using an adaptive regularizing prior, as in the previous section.

First, suppose you ignore the varying intercepts and just use the overall mean across all ponds, α, to make your predictions for each pond. A lot of data contributes to your estimate of α, and so it can be quite precise. However, your estimate of α is unlikely to exactly match the mean of any particular pond. As a result, the total sample mean underfits the data. This is the **COMPLETE POOLING** approach, pooling the data from all ponds to produce a single estimate that is applied to every pond. This sort of model is equivalent to assuming that the variation among ponds is zero—all ponds are identical.

Second, suppose you use the survival proportions for each pond to make predictions. This means using a separate intercept for each pond. The blue points in FIGURE 13.1 are this same kind of estimate. In each particular pond, quite little data contributes to each estimate,

and so these estimates are rather imprecise. This is particularly true of the smaller ponds, where less data goes into producing the estimates. As a consequence, the error of these estimates is high, and they are rather overfit to the data. Standard errors for each intercept can be very large, and in extreme cases, even infinite. These are sometimes called the **NO POOLING** estimates. No information is shared across ponds. It's like assuming that the variation among ponds is infinite, so nothing you learn from one pond helps you predict another.

Third, when you estimate varying intercepts, you use **PARTIAL POOLING** of information to produce estimates for each cluster that are less underfit than the grand mean and less overfit than the no-pooling estimates. As a consequence, they tend to be better estimates of the true per-cluster (per-pond) means. This will be especially true when ponds have few tadpoles in them, because then the no pooling estimates will be especially overfit. When a lot of data goes into each pond, then there will be less difference between the varying effect estimates and the no pooling estimates.

To demonstrate this fact, 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.

Learning to simulate and validate models and model fitting in this way is extremely valuable. Once you start using more complex models, you will want to ensure that your code is working and that you understand the model. You can help in this project by simulating data from the model, with specified parameter values, and then making sure that your method of estimation can recover the parameters within tolerable ranges of precision. Even just simulating data from a model structure has a huge impact on understanding.

13.2.1. The model. The first step is to define the model we'll be using. I'll use the same basic multilevel binomial model as before, but now with "ponds" instead of "tanks":

$$\begin{aligned} S_i &\sim \text{Binomial}(N_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{POND}[i]} \\ \alpha_j &\sim \text{Normal}(\bar{\alpha}, \sigma) \\ \bar{\alpha} &\sim \text{Normal}(0, 1.5) \\ \sigma &\sim \text{Exponential}(1) \end{aligned}$$

So to simulate data from this process, we need to assign values to:

- $\bar{\alpha}$, the average log-odds of survival in the entire population of ponds
- σ, the standard deviation of the distribution of log-odds of survival among ponds
- α, a vector of individual pond intercepts, one for each pond

We'll also need to assign sample sizes, N_i, to each pond. But once we've made all of those choices, we can easily simulate counts of surviving tadpoles, straight from the top-level binomial process, using `rbinom`. We'll do it all one step at a time.

Note that the priors are part of the model when we estimate, but not when we simulate. Why? Because priors are epistemology, not ontology. They represent the initial state of information of our robot, not a statement about how nature chooses parameter values.

13.2.2. Assign values to the parameters. I'm going to assign specific values representative of the actual tadpole data, to make the upcoming plot that demonstrates the increased accuracy of the varying effects estimates. But you can come back to this step later and change them to whatever you want.

Here's the code to initialize the values of α, σ, the number of ponds, and the sample size n_i in each pond.

R code 13.7

```
a_bar <- 1.5
sigma <- 1.5
nponds <- 60
Ni <- as.integer( rep( c(5,10,25,35) , each=15 ) )
```

I've chosen 60 ponds, with 15 each of initial tadpole density 5, 10, 25, and 35. I've chosen these densities to illustrate how the error in prediction varies with sample size. The use of `as.integer` in the last line arises from a subtle issue with how Stan, and therefore `ulam`, works. See the Overthinking box at the bottom of the page for an explanation.

The values $\bar{\alpha} = 1.4$ and $\sigma = 1.5$ define a Gaussian distribution of individual pond log-odds of survival. So now we need to simulate all 60 of these intercept values from the implied Gaussian distribution with mean $\bar{\alpha}$ and standard deviation σ:

R code 13.8

```
set.seed(5005)
a_pond <- rnorm( nponds , mean=a_bar , sd=sigma )
```

Go ahead and inspect the contents of `a_pond`. It should contain 60 log-odds values, one for each simulated pond.

Finally, let's bundle some of this information in a data frame, just to keep it organized.

R code 13.9

```
dsim <- data.frame( pond=1:nponds , Ni=Ni , true_a=a_pond )
```

Go ahead and inspect the contents of `dsim`, the simulated data. The first column is the pond index, 1 through 60. The second column is the initial tadpole count in each pond. The third column is the true log-odds survival for each pond.

Overthinking: Data types and Stan models. There are two basic types of numerical data in R, integers and real values. A number like "3" could be either. Inside your computer, integers and real ("`numeric`") values are represented differently. For example, here is the same vector of values generated as both:

R code 13.10

```
class(1:3)
class(c(1,2,3))
```

```
[1] "integer"
[1] "numeric"
```

Usually, you don't have to manage these types, because R manages them for you. But when you pass values to Stan, or another external program, often the internal representation does matter. In particular, Stan and `ulam` sometimes require explicit integers. For example, in a binomial model, the "size" variable that specifies the number of trials must be of integer type. Stan may provide a mysterious warning message about a function not being found, when the size variable is instead of "real" type, or what R calls `numeric`. Using `as.integer` before passing the data to Stan or `ulam` will resolve the issue.

13.2.3. Simulate survivors. Now we're ready to simulate the binomial survival process. 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 logistic function.

Putting the logistic into the random binomial function, we can generate a simulated survivor count for each pond:

R code 13.11

```
dsim$Si <- rbinom( nponds , prob=logistic(dsim$true_a) , size=dsim$Ni )
```

As usual with R, if you give it a list of values, it returns a new list of the same length. In the above, each paired α_i (`dsim$true_a`) and N_i (`dsim$Ni`) is used to generate a random survivor count with the appropriate probability of survival and maximum count. These counts are stored in a new column in `dsim`.

13.2.4. Compute the no-pooling estimates. We're ready to start analyzing the simulated data now. The easiest task is to just compute the no-pooling estimates. We can accomplish this straight from the empirical data, just by calculating the proportion of survivors in each pond. I'll keep these estimates on the probability scale, instead of translating them to the log-odds scale, because we'll want to compare the quality of the estimates on the probability scale later.

R code 13.12

```
dsim$p_nopool <- dsim$Si / dsim$Ni
```

Now there's another column in `dsim`, containing the empirical proportions of survivors in each pond. 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.

13.2.5. Compute the partial-pooling estimates. Now to fit the model to the simulated data, using `map2stan`. I'll use a single long chain in this example, but keep in mind that you need to use multiple chains to check convergence to the right posterior distribution. In this case, it's safe. But don't get cocky.

R code 13.13

```
dat <- list( Si=dsim$Si , Ni=dsim$Ni , pond=dsim$pond )
m13.3 <- ulam(
    alist(
        Si ~ dbinom( Ni , p ),
        logit(p) <- a_pond[pond],
        a_pond[pond] ~ dnorm( a_bar , sigma ),
        a_bar ~ dnorm( 0 , 1.5 ),
        sigma ~ dexp( 1 )
    ), data=dat , chains=4 )
```

We've fit the basic varying intercept model above. You can take a look at the estimates for $\bar{\alpha}$ and σ with the usual `precis` approach:

R code 13.14
```
precis( m13.3 , depth=2 )
```

```
            mean   sd  5.5% 94.5% n_eff Rhat
a_pond[1]   0.29 0.81 -0.97  1.59  3225 1.00
a_pond[2]   2.76 1.15  1.13  4.78  2050 1.00
...
a_pond[59]  1.87 0.46  1.17  2.66  3579 1.00
a_pond[60]  2.38 0.55  1.58  3.32  2829 1.00
a_bar       1.82 0.22  1.48  2.19  1706 1.00
sigma       1.41 0.21  1.11  1.78   708 1.01
```

I've abbreviated the output, since there are 60 intercept parameters, one for each pond.

Now let's compute the predicted survival proportions and add those proportions to our growing simulation data frame. To indicate that it contains the partial pooling estimates, I'll call the column `p_partpool`.

R code 13.15
```
post <- extract.samples( m13.3 )
dsim$p_partpool <- apply( inv_logit(post$a_pond) , 2 , mean )
```

If we want to compare to the true per-pond survival probabilities used to generate the data, then we'll also need to compute those, using the `true_a` column:

R code 13.16
```
dsim$p_true <- inv_logit( dsim$true_a )
```

The last thing we need to do, before we can plot the results and realize the point of this lesson, is to compute the absolute error between the estimates and the true varying effects. This is easy enough, using the existing columns:

R code 13.17
```
nopool_error <- abs( dsim$p_nopool - dsim$p_true )
partpool_error <- abs( dsim$p_partpool - dsim$p_true )
```

Now we're ready to plot. This is enough to get the basic display:

R code 13.18
```
plot( 1:60 , nopool_error , xlab="pond" , ylab="absolute error" ,
    col=rangi2 , pch=16 )
points( 1:60 , partpool_error )
```

I've decorated this plot with some additional information, displayed in FIGURE 13.3. The filled blue points in FIGURE 13.3 display the no-pooling estimates. The black circles show the varying effect estimates. The horizontal axis is the pond index, from 1 through 60. The vertical axis is the distance between the mean estimated probability of survival and the actual probability of survival. So points close to the bottom had low error, while those near the top had a large error, more than 20% off in some cases. The vertical lines divide the groups of ponds with different initial densities of tadpoles. And finally, the horizontal blue and black line segments show the average error of the no-pooling and partial pooling estimates, respectively, for each group of ponds with the same initial size. You can calculate these average error rates using `aggregate`:

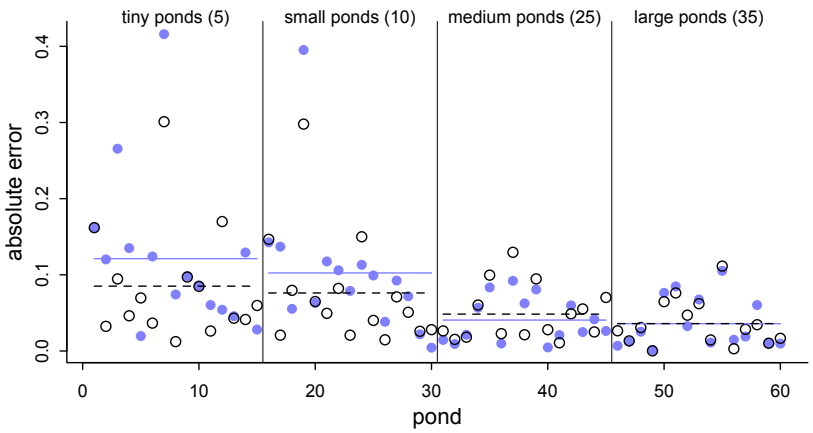

FIGURE 13.3. Error of no-pooling and partial pooling estimates, for the simulated tadpole ponds. The horizontal axis displays pond number. The vertical axis measures the absolute error in the predicted proportion of survivors, compared to the true value used in the simulation. The higher the point, the worse the estimate. No-pooling shown in blue. Partial pooling shown in black. The blue 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.

R code 13.19

```
nopool_avg <- aggregate(nopool_error,list(dsim$Ni),mean)
partpool_avg <- aggregate(partpool_error,list(dsim$Ni),mean)
```

The first thing to notice about FIGURE 13.3 plot is that both kinds of estimates are much more accurate for larger ponds, on the right side. This arises because more data means better estimates, assuming there is no confounding. If there is confounding, more data may just makes things worse. But there is no confounding in this simulated example. In the small ponds, sample size is small, and neither no-pooling nor partial-pooling can work magic. Therefore, prediction suffers on the left side of the plot. Second, note that the blue line is always above or very close to the black dashed line. This indicates that the no-pool estimates, shown by the blue points, have higher average error in each group of ponds, except for the medium ponds. Partial pooling isn't always better. It's just better on average in the long run. Even though both kinds of estimates get worse as sample size decreases, the varying effect estimates have the advantage, on average. Third, the distance between the blue line and the black dashed line grows as ponds get smaller. So while both kinds of estimates suffer from reduced sample size, the partial pooling estimates suffer less.

The pattern displayed in the figure is representative, but only one random simulation. To see how to quickly re-run the model on newly simulated data, without re-compiling the model, see the Overthinking box at the end of this section.

Okay, so what are we to make of all of this? Remember, back in FIGURE 13.1 (page 420), the smaller tanks demonstrated more shrinkage towards the mean. Here, the ponds with the smallest sample size show the greatest improvement over the naive no-pooling estimates. This is no coincidence. Shrinkage towards the mean results from trying to negotiate the underfitting and overfitting risks of the grand mean on one end and the individual means of each pond on the other. The smaller tanks/ponds contain less information, and so their varying estimates are influenced more by the pooled information from the other ponds. In other words, small ponds are prone to overfitting, and so they receive a bigger dose of the underfit grand mean. Likewise, the larger ponds shrink much less, because they contain more information and are prone to less overfitting. Therefore they need less correcting. When individual ponds are very large, pooling in this way does hardly anything to improve estimates, because the estimates don't have far to go. But in that case, they also don't do any harm, and the information pooled from them can substantially help prediction in smaller ponds.

The partially pooled estimates are better on average. They adjust individual cluster (pond) estimates to negotiate the trade-off between underfitting and overfitting. This is a form of regularization, just like in Chapter 7, but now with an amount of regularization that is learned from the data itself.

But there are some cases in which the no-pooling estimates are better. These exceptions often result from ponds with extreme probabilities of survival. The partial pooling estimates shrink such extreme ponds towards the mean, because few ponds exhibit such extreme behavior. But sometimes outliers really are outliers.

Overthinking: Repeating the pond simulation. This model samples pretty quickly. Compiling the model takes up most of the execution time. Luckily the compilation only has to be done once. Then you can pass new data to the compiled model and get new estimates. Once you've compiled `m13.3` once, you can use this code to re-simulate ponds and sample from the new posterior, without waiting for the model to compile again:

R code 13.20

```
a <- 1.5
sigma <- 1.5
nponds <- 60
Ni <- as.integer( rep( c(5,10,25,35) , each=15 ) )
a_pond <- rnorm( nponds , mean=a , sd=sigma )
dsim <- data.frame( pond=1:nponds , Ni=Ni , true_a=a_pond )
dsim$Si <- rbinom( nponds,prob=inv_logit( dsim$true_a ),size=dsim$Ni )
dsim$p_nopool <- dsim$Si / dsim$Ni
newdat <- list(Si=dsim$Si,Ni=dsim$Ni,pond=1:nponds)
m13.3new <- stan( fit=m13.3@stanfit , data=newdat , chains=4 )

post <- extract.samples( m13.3new )
dsim$p_partpool <- apply( inv_logit(post$a_pond) , 2 , mean )
dsim$p_true <- inv_logit( dsim$true_a )
nopool_error <- abs( dsim$p_nopool - dsim$p_true )
partpool_error <- abs( dsim$p_partpool - dsim$p_true )
plot( 1:60 , nopool_error , xlab="pond" , ylab="absolute error" , col=rangi2 , pch=16 )
points( 1:60 , partpool_error )
```

The `stan` function reuses the compiled model in `m13.3`, which is stored in the `stanfit` slot, passes it the new data, and returns the new samples in `m13.3new`. This is a useful trick, in case you want to perform a simulation study of a particular model structure.

13.3. More than one type of cluster

We can use and often should use more than one type of cluster in the same model. For example, the observations in `data(chimpanzees)`, which you met back in Chapter 11, are lever pulls. Each pull is within a cluster of pulls belonging to an individual chimpanzee. But each pull is also within an experimental block, which represents a collection of observations that happened on the same day. So each observed pull belongs to both an `actor` (1 to 7) and a `block` (1 to 6). There may be unique intercepts for each actor as well as for each block.

So in this section we'll reconsider the chimpanzees data, using both types of clusters simultaneously. This will allow us to use partial pooling on both categorical variables, `actor` and `block`, at the same time. We'll also get estimates of the variation among actors and among blocks.

Rethinking: Cross-classification and hierarchy. The kind of data structure in `data(chimpanzees)` is usually called a CROSS-CLASSIFIED multilevel model. It is cross-classified, because actors are not nested within unique blocks. If each chimpanzee had instead done all of his or her pulls on a single day, within a single block, then the data structure would instead be *hierarchical*. However, the model specification would typically be the same. So the model structure and code you'll see below will apply both to cross-classified designs and hierarchical designs. Other software sometimes forces you to treat these differently, on account of using a conditioning engine substantially less capable than MCMC. There are other types of "hierarchical" multilevel models, types that make adaptive priors for adaptive priors. It's turtles all the way down, recall (page 14). You'll see an example in the next chapter. But for the most part, people (or their software) nearly always use the same kind of model in both cases.

13.3.1. Multilevel chimpanzees.

Let's proceed by taking the chimpanzees model from Chapter 11 (`m11.4`, page 338) and add varying intercepts. To add varying intercepts to this model, we just replace the fixed regularizing prior with an adaptive prior. We'll also add a second cluster type. To add the second cluster type, `block`, we merely replicate the structure for the `actor` cluster. This means the linear model gets yet another varying intercept, $\alpha_{\text{BLOCK}[i]}$, and the model gets another adaptive prior and yet another standard deviation parameter.

Here is the mathematical form of the model, with the new pieces of the machine highlighted in blue:

$$\begin{aligned}
L_i &\sim \text{Binomial}(1, p_i) \\
\text{logit}(p_i) &= \alpha_{\text{ACTOR}[i]} + \gamma_{\text{BLOCK}[i]} + \beta_{\text{TREATMENT}[i]} \\
\beta_j &\sim \text{Normal}(0, 0.5) \quad \text{, for } j = 1..4 \\
\alpha_j &\sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \quad \text{, for } j = 1..7 \\
\gamma_j &\sim \text{Normal}(0, \sigma_\gamma) \quad \text{, for } j = 1..6 \\
\bar{\alpha} &\sim \text{Normal}(0, 1.5) \\
\sigma_\alpha &\sim \text{Exponential}(1) \\
\sigma_\gamma &\sim \text{Exponential}(1)
\end{aligned}$$

Each cluster gets its own vector of parameters. For actors, the vector is α, and it has length 7, because there are 7 chimpanzees in the sample. For blocks, the vector is γ, and it has length 6, because there are 6 blocks. Each cluster variable needs its own standard deviation parameter

that adapts the amount of pooling across units, be they actors or blocks. These are σ_α and σ_γ, respectively. Finally, note that there is only one global mean parameter $\bar{\alpha}$. We can't identify a separate mean for each varying intercept type, because both intercepts are added to the same linear prediction. If you do include a mean for each cluster type, it won't be the end of the world, however. It'll be like the right leg and left leg example from Chapter 6.

Now to run the model that uses both `actor` and `block`:

R code 13.21

```
library(rethinking)
data(chimpanzees)
d <- chimpanzees
d$treatment <- 1 + d$prosoc_left + 2*d$condition

dat_list <- list(
    pulled_left = d$pulled_left,
    actor = d$actor,
    block_id = d$block,
    treatment = as.integer(d$treatment) )

set.seed(13)
m13.4 <- ulam(
    alist(
        pulled_left ~ dbinom( 1 , p ) ,
        logit(p) <- a[actor] + g[block_id] + b[treatment] ,
        b[treatment] ~ dnorm( 0 , 0.5 ),
      ## adaptive priors
        a[actor] ~ dnorm( a_bar , sigma_a ),
        g[block_id] ~ dnorm( 0 , sigma_g ),
      ## hyper-priors
        a_bar ~ dnorm( 0 , 1.5 ),
        sigma_a ~ dexp(1),
        sigma_g ~ dexp(1)
    ) , data=dat_list , chains=4 , cores=4 , log_lik=TRUE )
```

You'll end up with 2000 samples from 4 independent chains. As always, be sure to inspect the trace plots and the diagnostics. As soon as you start trusting the machine, the machine will betray your trust. In this case, you should see a warning about **DIVERGENT TRANSITIONS**:

```
Warning messages:
1: There were 22 divergent transitions after warmup.
```

The model did actually sample fine. But these warnings indicate that it had some trouble efficiently exploring the posterior. In the next section, I'll show you how to fix this. For now, we can keep moving and interpret the posterior.

This is easily the most complicated model we've used in the book so far. So let's look at the posterior and take note of a few important features:

R code 13.22

```
precis( m13.4 , depth=2 )
plot( precis(m13.4,depth=2) ) # also plot
```

```
         mean   sd  5.5% 94.5% n_eff Rhat
b[1]    -0.12 0.30 -0.59  0.39   158 1.03
```

```
b[2]     0.40 0.30 -0.07  0.88   310 1.02
b[3]    -0.48 0.30 -0.96  0.00   515 1.01
b[4]     0.30 0.31 -0.17  0.80   186 1.02
a[1]    -0.37 0.36 -0.94  0.24   446 1.01
a[2]     4.61 1.20  2.98  6.83   915 1.01
a[3]    -0.67 0.36 -1.24 -0.08   709 1.01
a[4]    -0.68 0.37 -1.26 -0.09   235 1.02
a[5]    -0.37 0.36 -0.93  0.19   338 1.01
a[6]     0.57 0.35  0.01  1.12   560 1.01
a[7]     2.09 0.45  1.41  2.82   721 1.01
g[1]    -0.17 0.22 -0.57  0.07   426 1.01
g[2]     0.05 0.18 -0.19  0.36   921 1.01
g[3]     0.05 0.19 -0.22  0.39  1062 1.01
g[4]     0.02 0.18 -0.25  0.31   939 1.01
g[5]    -0.02 0.18 -0.31  0.24   873 1.00
g[6]     0.12 0.19 -0.11  0.49   533 1.01
a_bar    0.58 0.74 -0.58  1.79   800 1.00
sigma_a  2.00 0.66  1.17  3.16  1106 1.00
sigma_g  0.21 0.17  0.03  0.52   229 1.02
```

The `precis` plot is shown in the left-hand part of FIGURE 13.4 (page 432).

First, notice that the number of effective samples, `n_eff`, varies quite a lot across parameters. This is common in complex models. Why? There are many reasons for this. But in this sort of model a common reason is that some parameter spends a lot of time near a boundary. Here, that parameter is `sigma_g`. It spends a lot of time near its minimum of zero. Some Rhat values are also slightly above 1.00 now. All of this is a sign of inefficient sampling, which we'll fix in the next section.

Second, compare `sigma_a` to `sigma_g` and notice that the estimated variation among actors is a lot larger than the estimated variation among blocks. This is easy to appreciate, if we plot the marginal posterior distributions of these two parameters. I've down this on the right in FIGURE 13.4. While there's uncertainty about the variation among actors, this model is confident that actors vary more than blocks. You can easily see this variation in the varying intercept distributions: the `a` distributions are much more scattered than are the `g` distributions. The chimpanzees vary, but the blocks are all the same.

As a consequence, adding `block` to this model hasn't added a lot of overfitting risk. Let's compare the model with only varying intercepts on `actor` to the model with both kinds of varying intercepts. The model that ignores block is:

R code 13.23

```
set.seed(14)
m13.5 <- ulam(
    alist(
        pulled_left ~ dbinom( 1 , p ) ,
        logit(p) <- a[actor] + b[treatment] ,
        b[treatment] ~ dnorm( 0 , 0.5 ),
        a[actor] ~ dnorm( a_bar , sigma_a ),
        a_bar ~ dnorm( 0 , 1.5 ),
        sigma_a ~ dexp(1)
    ) , data=dat_list , chains=4 , cores=4 , log_lik=TRUE )
```

Comparing to the model with both clusters:

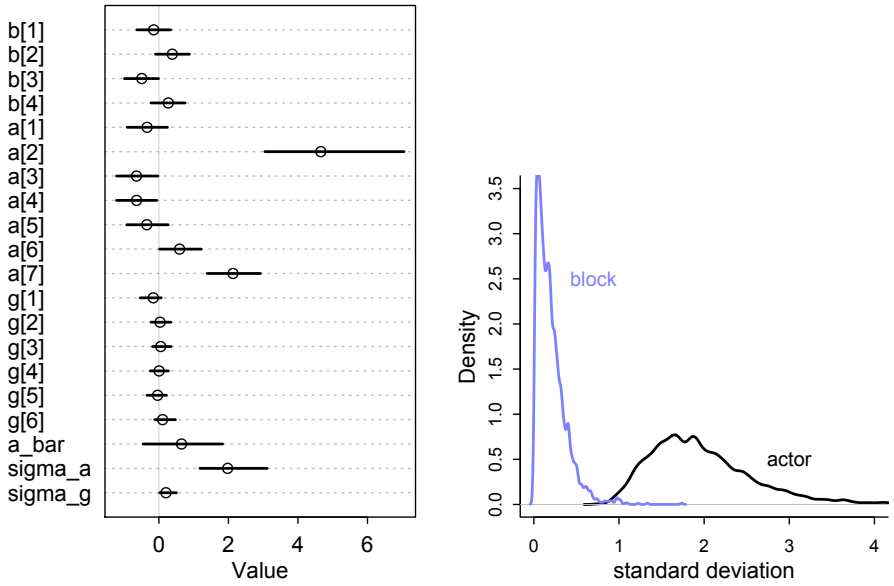

FIGURE 13.4. Left: Posterior means and 89% compatibility intervals for `m13.4`. The greater variation across actors than blocks can be seen immediately in the `a` and `g` distributions. Right: Posterior distributions of the standard deviations of varying intercepts by actor (black) and block (blue).

R code 13.24

```
compare( m13.4 , m13.5 )
```

```
       WAIC    SE dWAIC  dSE pWAIC weight
m13.5 531.3 19.25     0   NA   8.6   0.63
m13.4 532.3 19.33     1 1.71  10.7   0.37
```

Look at the `pWAIC` column, which reports the "effective number of parameters." While `m13.4` has 7 more parameters than `m13.5` does, it has only 2 more effective parameters. Why? Because the posterior distribution for `sigma_g` ended up close to zero. This means each of the 6 `g` parameters is strongly shrunk towards zero—they are relatively inflexible. In contrast, the `a` parameters are shrunk towards zero much less, because the estimated variation across actors is much larger, resulting in less shrinkage. But as a consequence, each of the `a` parameters contributes much more to the `pWAIC` value.

You might also notice that the difference in WAIC between these models is small, only about 1. This is especially small compared the standard error of the difference. These two models imply nearly identical predictions, and so their expected out-of-sample accuracy is nearly identical. The block parameters have been shrunk so much towards zero that they do very little work in the model.

If you are feeling the urge to "select" `m13.4` as the best model, pause for a moment. There is nothing to gain here by selecting either model. The comparison of the two models tells a richer story—whether we include block or not hardly matters, and the `g` and `sigma_g`

estimates tell us why. By retaining and reporting both models, we and our readers learn more about the experiment. Model comparison is of value. To select a model, we'd rather want to test conditional independencies of different causal models. Since this is an experiment, there is nothing to really select. The experimental design tells us the relevant causal model to inspect.

13.3.2. Even more clusters. You might notice that the treatment effects, the `b` parameters, look a lot like the `a` and `g` parameters. Could we also use partial pooling on the treatment effects? Yes, we could. Some people will scream "No!" at this suggestion, because they have been taught that varying effects are only for variables that were not experimentally controlled. Since treatment was "fixed" by the experiment, the thinking goes, we should use un-pooled "fixed" effects.

This is all wrong. The reason to use varying effects is because they provide better inferences. It doesn't matter how the clusters arise. If the individual units are **EXCHANGABLE**—the index values could be reassigned without changing the meaning of the model—then partial pooling could help.

In this case, there are only four treatments and there is a lot of data on each treatment. So partial pooling isn't going to make any difference anyway. Here is `m13.4` but now with partial pooling on the treatments:

R code 13.25

```
set.seed(15)
m13.6 <- ulam(
    alist(
        pulled_left ~ dbinom( 1 , p ) ,
        logit(p) <- a[actor] + g[block_id] + b[treatment] ,
        b[treatment] ~ dnorm( 0 , sigma_b ),
        a[actor] ~ dnorm( a_bar , sigma_a ),
        g[block_id] ~ dnorm( 0 , sigma_g ),
        a_bar ~ dnorm( 0 , 1.5 ),
        sigma_a ~ dexp(1),
        sigma_g ~ dexp(1),
        sigma_b ~ dexp(1)
    ) , data=dat_list , chains=4 , cores=4 , log_lik=TRUE )
coeftab( m13.4 , m13.6 )
```

```
        m13.4   m13.6
b[1]      -0.13   -0.14
b[2]       0.39    0.35
b[3]      -0.48   -0.47
b[4]       0.28    0.24
```

I cut off the rest of the `coeftab` output. We're only interested in the `b` parameters right now. These are not identical, but they are very close. If you look at `sigma_b`, you'll that it is small. The treatments don't vary a lot, on the logit scale, because they don't make much difference in the first place. And there is a lot of data in each treatment, so they don't get pooled much in any event. If you compare model `m13.6` with `m13.4`, using either WAIC or PSIS, you'll see they are no different on purely predictive criteria. This is the typical result, when each cluster (each treatment here) has a lot of data to inform its parameters.

What you do get from `m13.6` are more **DIVERGENT TRANSITIONS**. So let's finally deal with those.

13.4. Divergent transitions and non-centered priors

With the models in the previous section, Stan reported warnings about **DIVERGENT TRANSITIONS**. You first heard about these back in Chapter 9, and I promised to explain them later. Now is the time to learn what these things are and a few useful ways to fix them. When you work with multilevel models, divergent transitions are commonplace. So you need to know how to fix them, and that requires knowing something about what causes them.

One of the best things about Hamiltonian Monte Carlo is that it provides internal checks of efficiency and accuracy. One of these checks comes free, arising from the constraints on the physics simulation. Recall that HMC simulates the frictionless flow of a particle on a surface. In any given transition, which is just a single flick of the particle, the total energy at the start should be equal to the total energy at the end. That's how energy in a closed system works. And in a purely mathematical system, the energy is always conserved correctly. It's just a fact about the physics.

But in a numerical system, it might not be. Sometimes the total energy is not the same at the end as it was at the start. In these cases, the energy is *divergent*. How can this happen? It tends to happen when the posterior distribution is very steep in some region of parameter space. Steep changes in probability are hard for a discrete physics simulation to follow. When that happens, the algorithm notices by comparing the energy at the start to the energy at the end. When they don't match, it indicates numerical problems exploring that part of the posterior distribution.

Divergent transitions are rejected. They don't directly damage your approximation of the posterior distribution. But they do hurt it indirectly, because the region where divergent transitions happen is hard to explore correctly. And even when there aren't any divergent transitions, distributions with steep regions are hard to explore. The chains will be less efficient. And unfortunately this happens quite often in multilevel models.

There are two easy tricks for reducing the impact of divergent transitions. The first is to tune the simulation so that it doesn't overshoot the valley wall. This means doing more warmup with a higher target acceptance rate, Stan's `adapt_delta`. But for many models, you can never tune the sampler enough to remove the divergent transitions. The second trick is to write the statistical model in a new way, to **REPARAMETERIZE** it. For any given statistical model, it can be written in several forms that are mathematically identical but numerically different. Switching a model from one form to another is called reparameterization. Let's work through two examples.

Rethinking: No free samples. When Hamiltonian Monte Carlo complains about divergent transitions, it is tempting to fall back on some other sampler that complains less. This is a mistake. A Gibbs sampler, for example, will never complain. It will just silently fail. It is true that Gibbs sampling doesn't have the same problem with steep curvature that HMC has. But Gibbs still has problems with the same posterior distributions. It just provides no warnings.

The general issue—warnings of unreliable approximations—arises in all parts of computational statistics. The R package `lme4` is a nice package for fitting multilevel models. It isn't Bayesian, but instead uses a clever non-Bayesian algorithm. Sometimes that algorithm is unreliable, and `lme4` is very good about warning the user. Alternative packages that try to fit the same multilevel models may not produce warnings nearly as often. But those packages are no more reliable. They are just less cautious.

13.4.1. The Devil's Funnel. You don't need a fancy model to experience divergent transitions. Suppose we have this joint distribution of two variables, v and x:

$$v \sim \text{Normal}(0, 3)$$
$$x \sim \text{Normal}(0, \exp(v))$$

There are no data here, just a joint distribution to sample from. This distribution might seem weird, but it represents a typical multilevel distribution, in which the scale of one variable (here x) depends upon another variable (here v). We'll visualize it on the next page. You can try this in `ulam()`:

R code 13.26

```
m13.7 <- ulam(
    alist(
        v ~ normal(0,3),
        x ~ normal(0,exp(v))
    ), data=list(N=1) , chains=4 )
precis( m13.7 )
```

```
   mean     sd   5.5%  94.5% n_eff Rhat
v  1.90   2.08  -1.49   5.42    39 1.06
x 18.12 135.97 -31.78 123.84   102 1.04
```

This looks like an easy problem—only two parameters—but it's a disaster. You should see lots of divergent transitions. And the `n_eff` and `Rhat` values are very poor. Take a glance at the trace plot, `traceplot(m13.7)`, too.

This example is The Devil's Funnel.[192] In the left panel of Figure 13.5, I show the distribution's contours. At low values of v, the distribution of x contracts around zero. This forms a very steep valley that the Hamiltonian particle needs to explore. Steep surfaces are hard to simulate, because the simulation is not actually continuous. It happens in discrete steps. If the steps are too big, the simulation will overshoot. This error effectively changes the total energy in the system. What happens next is unpredictable.

As in the examples in Chapter 9, the simulation in Figure 13.5 (left panel) starts at the $\times$. The simulation finds the valley. But then it misses its turn and careens into space. The open point is a divergent transition, a proposal for which the energy at the start of the transition is not the same as the energy at the end of the transition. When you try to sample from this distribution, you get lots of these divergent transitions and a very unreliable approximation of the posterior distribution. We can prove that in this case, because it is a very simple distribution that we can compute with grid approximation.

We can fix this problem by reparameterizing the funnel. There are two general ways to parameterize models in which the distribution of one parameter is a function of another parameter. In this example, the distribution of x is a function of v:

$$x \sim \text{Normal}(0, \exp(v))$$

This is the source of the funnel: As v changes, the distribution of x changes in a very inconvenient way. This parameterization is known as the **centered parameterization**. This is not a very intuitive name. It just indicates that the distribution of x is conditional on one or more other parameters.

The alternative is a **non-centered parameterization**. A non-centered parameterization moves the embedded parameter, v in this case, out of the definition of the other

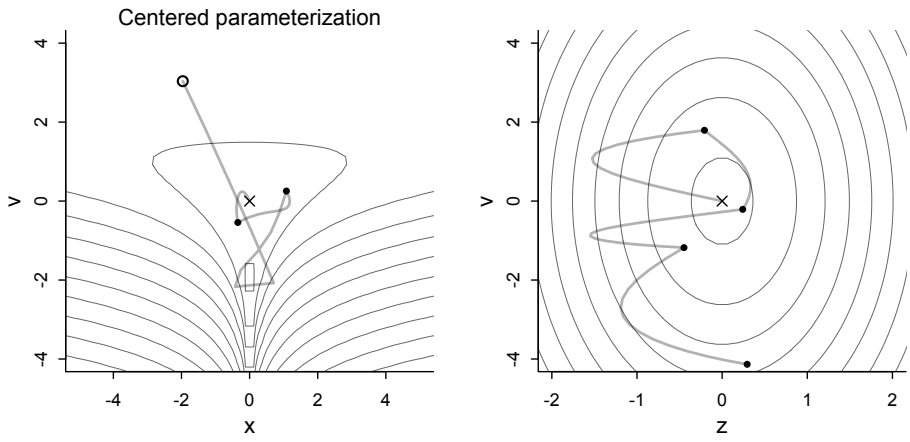

FIGURE 13.5. Divergent transitions happen when the posterior is steep and the HMC simulation is too coarse to follow it. These numerical errors are detected automatically. Left: The posterior distribution here is a steep valley around $x = 0$ when v is small. The divergent transition (open point) overshoots the wall of the valley and then careens wildly into space. Right: The same model, but with a non-centered parameterization that flattens the valley. See the model definitions in the text. See examples in `?HMC_2D_sample` for code to reproduce these figures.

parameter. For The Devil's Funnel, we can accomplish that like this:

$$\begin{aligned} v &\sim \text{Normal}(0, 3) \\ z &\sim \text{Normal}(0, 1) \\ x &= z \exp(v) \end{aligned}$$

This looks crazy. So to understand what just happened, consider the common procedure of standardizing a variable. Many times so far in this book, we've standardized data before running a model. The procedure is to subtract the mean and then divide by the standard deviation. The new, standardized variable has mean zero and standard deviation one. To get the original variable back, you would perform these steps in reverse. First you'd multiply the standardized variable by the original standard deviation. Then you'd add the original mean.

The reparameterization above has just defined z as the standardized x. Since it is standardized, it has mean zero and standard deviation one. Then to compute x, we reverse the standardization by multiplying z by the standard deviation, $\exp(v)$. There is no mean to add back, because the mean in both cases is zero. But if there were a different mean, we'd add it back in this step as well. The result is that x in the non-centered version has the same distribution as x in the original, centered version. It's the same joint distribution of v and x.

But when we run the Markov chain, it's rather different. We don't sample x directly now. Instead we sample z. The righthand panel of FIGURE 13.5 shows the non-centered distribution's contours—it's just a bivariate Gaussian now—and the HMC simulation on top. Let's run the model again in `ulam()`:

R code 13.27

```
m13.7nc <- ulam(
    alist(
        v ~ normal(0,3),
        z ~ normal(0,1),
        gq> real[1]:x <<- z*exp(v)
    ), data=list(N=1) , chains=4 )
precis( m13.7nc )
```

```
   mean     sd   5.5% 94.5% n_eff Rhat
v -0.04   2.88  -4.63  4.58  1612    1
z  0.01   0.99  -1.57  1.62  1555    1
x -3.70 260.03 -25.35 23.12  1511    1
```

All is well. If you plot x against v, you will see the funnel. We managed to sample it by sampling a different variable and then transforming it. That is the non-centered parameterization. It's used often when working with multilevel models. However, there are times when the centered prior is better. So it pays to be comfortable with both.

13.4.2. Non-centered chimpanzees. For a real example, Llet's return to the chimpanzees. In model `m13.4`, the adaptive priors that make it a multilevel model have parameters inside them. These are causing regions of steep curvature and generating divergent transitions. We can fix that though.

Before reparameterizing, the first thing you can try is to increase Stan's target acceptance rate. This is controlled by the `adapt_delta` control parameter. The `ulam` default is 0.95, which means that it aims to attain a 95% acceptance rate. It tries this during the warmup phase, adjusting the step size of each leapfrog step (go back to Chapter 9 if these terms aren't familiar). When `adapt_delta` is set high, it results in a smaller step size, which means a more accurate approximation of the curved surface. It can also mean slower exploration of the distribution.

Increasing `adapt_delta` will often, but not always, help with divergent transitions. For example, model `m13.4` in the previous section presented a few divergent transitions. We can re-run the model, using a higher target acceptance rate, with:

R code 13.28

```
set.seed(13)
m13.4b <- ulam( m13.4 , chains=4 , cores=4 , control=list(adapt_delta=0.99) )
divergent(m13.4b)
```

```
[1] 2
```

So that did help. But sometimes this won't be enough. And while the divergences are gone, the chain still isn't very efficient—look at the `precis` output and notice that many of the `n_eff` values are still far below the true number of samples (2000 in this case: 4 chains, 500 from each).

We can do much better with the non-centered version of the model. What we want is a version of `m13.4` (page 429) in which we get the parameters out of the adaptive priors and instead into the linear model. There are two adaptive priors to transform:

$$\alpha_j \sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \quad \text{[Intercepts for actors]}$$

$$\gamma_j \sim \text{Normal}(0, \sigma_\gamma) \quad \text{[Intercepts for blocks]}$$

There are three embedded ("centered") parameters to smuggle out of these priors: $\bar{\alpha}$, σ_α, σ_γ. As before with the funnel, we'll define some new variables that are given standard Normal distributions, and then we'll reconstruct the original variables by undoing the transformation. This time, we'll do that reconstruction in the linear model. The completed non-centered model looks like this (with altered bits in blue):

$$\begin{aligned}
L_i &\sim \text{Binomial}(1, p_i) \\
\text{logit}(p_i) &= \underbrace{\bar{\alpha} + z_{\text{ACTOR}[i]}\sigma_\alpha}_{\alpha_{\text{ACTOR}[i]}} + \underbrace{x_{\text{BLOCK}[i]}\sigma_\gamma}_{\gamma_{\text{BLOCK}[i]}} + \beta_{\text{TREATMENT}[i]} \\
\beta_j &\sim \text{Normal}(0, 0.5) \quad \text{, for } j = 1..4 \\
z_j &\sim \text{Normal}(0, 1) \qquad \text{[Standardized actor intercepts]} \\
x_j &\sim \text{Normal}(0, 1) \qquad \text{[Standardized block intercepts]} \\
\bar{\alpha} &\sim \text{Normal}(0, 1.5) \\
\sigma_\alpha &\sim \text{Exponential}(1) \\
\sigma_\gamma &\sim \text{Exponential}(1)
\end{aligned}$$

The vector z gives the standardized intercept for each actor, and the vector x gives the standardized intercept for each block. Inside the linear model $\text{logit}(p_i)$, all of the previously embedded parameters reappear. Each actor intercept is defined by

$$\alpha_j = \bar{\alpha} + z_j\sigma_\alpha$$

and each block intercept by

$$\gamma_j = x_j\sigma_\gamma$$

So these expressions appear now in the linear model.

Let's sample from this posterior now and see what the reparameterization gains us.

R code 13.29

```
set.seed(13)
m13.4nc <- ulam(
    alist(
        pulled_left ~ dbinom( 1 , p ) ,
        logit(p) <- a_bar + z[actor]*sigma_a + # actor intercepts
                    x[block_id]*sigma_g +      # block intercepts
                    b[treatment] ,
        b[treatment] ~ dnorm( 0 , 0.5 ),
        z[actor] ~ dnorm( 0 , 1 ),
        x[block_id] ~ dnorm( 0 , 1 ),
        a_bar ~ dnorm( 0 , 1.5 ),
        sigma_a ~ dexp(1),
        sigma_g ~ dexp(1),
        gq> vector[actor]:a <<- a_bar + z*sigma_a,
        gq> vector[block_id]:g <<- x*sigma_g
    ) , data=dat_list , chains=4 , cores=4 )
```

Now let's compare the `n_eff`, numbers of effective samples, for these two forms. To do this fairly, we should ignore the `z` and `x` parameters and instead compare `a` and `g` parameters. That is why I added those `gq>` lines at the bottom of the formula above, so that Stan would

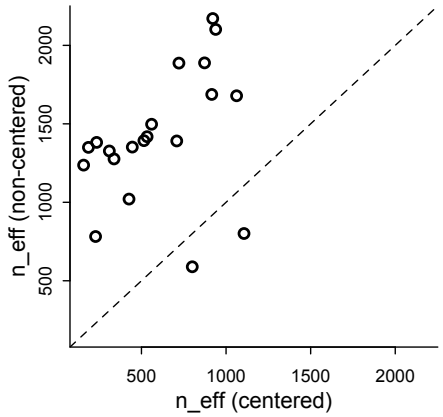

FIGURE 13.6. Comparing the centered (horizonal) and non-centered (vertical) parameterizations of the multilevel chimpanzees model, `m13.4`. Each point is a parameter. All but two parameters lie above the diagonal, indicating better sampling for the non-centered parameterization.

do the calculations for us while it ran. The code below pulls the matching `n_eff` values out of the `precis` tables for both models. Then it plots them against one another.

R code 13.30

```
precis_c <- precis( m13.4 , depth=2 )
precis_nc <- precis( m13.4nc , depth=2 )
pars <- c( paste("a[",1:7,"]",sep="") , paste("g[",1:6,"]",sep="") ,
           paste("b[",1:4,"]",sep="") , "a_bar" , "sigma_a" , "sigma_g" )
neff_table <- cbind( precis_c[pars,"n_eff"] , precis_nc[pars,"n_eff"] )
plot( neff_table , xlim=range(neff_table) , ylim=range(neff_table) ,
    xlab="n_eff (centered)" , ylab="n_eff (non-centered)" , lwd=2 )
abline( a=0 , b=1 , lty=2 )
```

The result is displayed in FIGURE 13.6. The diagonal shows where both models produce the same effective number of samples. For all but two parameters, the non-centered parameterization performs much better.

So should we always use the non-centered parameterization? No. Sometimes the centered form is better. It could even be true that the centered form is better for one cluster in a model while the non-centered form is better for another cluster in the same model. It all depends upon the details. Typically, a cluster with low variation, like the blocks in `m13.4`, will sample better with a non-centered prior. And if you have a large number of units inside a cluster, but not much data for each unit, then the non-centered is also usually better. But being able to switch back and forth as needed is very useful.

We can reparameterize distributions other than the Gaussian. For example, an exponential distribution has a single scale parameter, usually called λ, that can be factored out and smuggled into a linear model:

$$x = z\lambda$$
$$z \sim \text{Exponential}(1)$$

This is the same as $x \sim \text{Exponential}(\lambda)$. And in the next chapter, I'll show you how to reparameterize multivariate distributions so to place an entire correlation matrix inside a linear model. Algebra makes many things possible.

13.5. Multilevel posterior predictions

Way back in Chapter 3 (page 64), I commented on the importance of **MODEL CHECKING**. Software does not always work as expected, and one robust way to discover mistakes is to compare the sample to the posterior predictions of a fit model. The same procedure, 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.

Once you believe the posterior is correct, implied predictions are needed to consider the causal effects. What is the estimated effect of intervening on one or more variables? We need counterfactual posterior predictions for this question. We saw an example of this in Chapter 5.

Another role for constructing implied predictions is in computing **INFORMATION CRITERIA**, like AIC and WAIC. These criteria provide simple estimates of out-of-sample model accuracy, the KL divergence. In practical terms, information criteria provide a rough measure of a model's flexibility and therefore overfitting risk. This was the big conceptual mission of Chapter 7.

All of this advice applies to multilevel models as well. We still often need model checks, counterfactual predictions for understanding, and information criteria. 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. That is what shrinkage does for us. Of course, we should never be trying to really retrodict the sample. But now you have to expect that even a perfectly good model fit will differ from the raw data in a systematic way.

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 ones 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.

13.5.1. Posterior prediction for same clusters.

When working with the same clusters as you used to fit a model, varying intercepts are just parameters. The only trick is to ensure that you use the right intercept for each case in the data. If you use `link` and `sim` to do your work for you, this is handled automatically. Otherwise, you just use the model definition.

For example, in `data(chimpanzees)`, there are 7 unique actors. These are the clusters. The varying intercepts model, `m13.4`, estimated an intercept for each, in addition to two parameters to describe the mean and standard deviation of the population of actors. We'll construct posterior predictions (retrodictions), using both the automated `link` approach and doing it from scratch, so there is no confusion.

Before computing predictions, note again that we should no longer expect the posterior predictive distribution to match the raw data, even when the model worked correctly. Why? The whole point of partial pooling is to shrink estimates towards the grand mean. So the estimates should not necessarily match up with the raw data, once you use pooling.

The code needed to compute posterior predictions is just like the code from Chapter 11. Here it is again, computing and plotting posterior predictions for actor number 2:

R code 13.31

```
chimp <- 2
d_pred <- list(
    actor = rep(chimp,4),
    treatment = 1:4,
    block_id = rep(1,4)
)
p <- link( m13.4 , data=d_pred )
p_mu <- apply( p , 2 , mean )
p_ci <- apply( p , 2 , PI )
```

And the plotting code is exactly the same as before (page **??**).

To construct the same calculations without using `link`, we just have to remember the model. The only difficulty is that when we work with the samples from the posterior, the varying intercepts will be a matrix of samples. Let's take a look:

R code 13.32

```
post <- extract.samples(m13.4)
str(post)
```

```
List of 6
 $ b      : num [1:2000, 1:4] -0.107 -0.491 -0.644 -0.368 0.105 ...
 $ a      : num [1:2000, 1:7] -0.0166 -0.2078 0.3102 0.1337 -0.191 ...
 $ g      : num [1:2000, 1:6] -0.7116 -0.1728 -0.5689 -0.0299 0.0133 ...
 $ a_bar  : num [1:2000(1d)] 1.2031 -0.0998 1.3569 0.6167 -0.0248 ...
 $ sigma_a: num [1:2000(1d)] 3.1 3.57 2.92 2.15 2.19 ...
 $ sigma_g: num [1:2000(1d)] 0.393 0.287 0.418 0.119 0.13 ...
```

The `a` matrix has samples on the rows and actors on the columns. So to plot, for example, the density for actor 5:

R code 13.33

```
dens( post$a[,5] )
```

The `[,5]` means "all samples for actor 5."

To construct posterior predictions, we build our own link function. I'll use the `with` function here, so we don't have to keep typing `post$` before every parameter name:

R code 13.34

```
p_link <- function( treatment , actor=1 , block_id=1 ) {
    logodds <- with( post ,
        a[,actor] + g[,block_id] + b[,treatment] )
    return( inv_logit(logodds) )
}
```

The linear model is identical to the one used to define the model, but with a single comma added inside the brackets after `a`. Now to compute predictions:

R code 13.35

```
p_raw <- sapply( 1:4 , function(i) p_link( i , actor=2 , block_id=1 ) )
p_mu <- apply( p_raw , 2 , mean )
p_ci <- apply( p_raw , 2 , PI )
```

At some point, you will have to work with a model that `link` will mangle. At that time, you can return to this section and peer hard at the code above and still make progress. No matter what the model is, if it is a Bayesian model, then it is *generative*. This means that predictions are made by pushing samples up through the model to get distributions of predictions. Then you summarize the distributions to summarize the predictions.

13.5.2. Posterior prediction for new clusters. The problem of making predictions for new clusters is really a problem of generalizing from the sample. In general, there is no unique procedure for generalizing predictions outside of a sample. The right thing to do depends upon the causal model, the statistical model, and your goals. But if you have a generative model, then you can often think your way through it. The key idea is to use the posterior to parameterize a simulation that embodies the target generalization.

Let's consider some simple examples.

Suppose you want to predict how chimpanzees in another population would respond to our lever pulling experiment. The particular 7 chimpanzees in the sample allowed us to estimate 7 unique intercepts. But these individual actor intercepts aren't of interest, because none of these 7 individuals is in the new population.

One way to grasp the task of constructing posterior predictions for new clusters is to imagine leaving out one of the clusters when you fit the model to the data. For example, suppose we leave out actor number 7 when we fit the chimpanzees model. Now how can we assess the model's accuracy for predicting actor number 7's behavior? We can't use any of the `a` parameter estimates, because those apply to other individuals. But we can make good use of the `a_bar` and `sigma_a` parameters. These parameters describe a statistical population of actors, and we can simulate new actors from it.

First, let's see how to construct posterior predictions for a new, previously unobserved *average* actor. By "average," I mean an individual chimpanzee with an intercept exactly at `a_bar` ($\bar{\alpha}$), the population mean. Since there is uncertainty about the population mean, there is still uncertainty about this average individual's intercept. But as you'll see, the uncertainty is much smaller than it really should be, if we wish to honestly represent the problem of what to expect from a new individual.

What we need is our own link function, but now with twist:

R code 13.36

```
p_link_abar <- function( treatment ) {
    logodds <- with( post , a_bar + b[,treatment] )
    return( inv_logit(logodds) )
}
```

Notice that the function ignores `block`. This is because we are extrapolating to new blocks, so we assume the average block effect is about zero (which it was in the sample). Call this function and summarize just as before:

R code 13.37

```
post <- extract.samples(m13.4)
p_raw <- sapply( 1:4 , function(i) p_link_abar( i ) )
p_mu <- apply( p_raw , 2 , mean )
p_ci <- apply( p_raw , 2 , PI )

plot( NULL , xlab="treatment" , ylab="proportion pulled left" ,
    ylim=c(0,1) , xaxt="n" , xlim=c(1,4) )
```

```
axis( 1 , at=1:4 , labels=c("R/N","L/N","R/P","L/P") )
lines( 1:4 , p_mu )
shade( p_ci , 1:4 )
```

The result is displayed in FIGURE 13.7, on the left. The gray region shows the 89% compatibility interval for an actor with an average intercept. This kind of calculation makes it easy to see the impact of `prosoc_left`, as well as uncertainty about where the average is, but it doesn't show the variation among actors.

To show the variation among actors, we'll need to use `sigma_a` in the calculation. First we simply use `rnorm` to sample some random chimpanzees, using mean `a_bar` and standard deviation `sigma_a`. Then we write a link function that references those simulated chimpanzees, not the ones in the posterior. It's important to do the chimpanzee sampling outside the link function, because we want to reference the same simulate chimpanzee, whichever treatment we consider. This is the code:

R code 13.38

```
a_sim <- with( post , rnorm( length(post$a_bar) , a_bar , sigma_a ) )
p_link_asim <- function( treatment ) {
    logodds <- with( post , a_sim + b[,treatment] )
    return( inv_logit(logodds) )
}
p_raw_asim <- sapply( 1:4 , function(i) p_link_asim( i ) )
```

Summarizing and plotting is exactly as before, and the result is displayed in the middle of FIGURE 13.7. These posterior predictions are *marginal* of actor, which means that they average over the uncertainty among actors. In contrast, the predictions on the left just set the actor to the average, ignoring variation among actors.

At this point, students usually ask, "So which one should I use?" The answer is, "It depends." Both are useful, depending upon the question. The predictions for an average actor help to visualize the impact of treatment. The predictions that are marginal of actor illustrate how variable different chimpanzees are, according to the model. You probably want to compute both for yourself, when trying to understand a model. But which you include in a report will depend upon context.

In this case, we can do better by making a plot that displays both the treatment effect and the variation among actors. We can do this by forgetting about intervals and instead simulating a series of new actors in each of the four treatments. By drawing a line for each actor across all four treatments, we'll be able to visualize both the zig-zag impact of `prosoc_left` as well as the variation among individuals.

We don't really need new code here. We just need to use the rows in `p_raw_asim` from above. Each row contains a single trend, a single simulated chimpanzee. So instead of summarizing with `mean` and `PI`, we can just loop over rows and plot:

R code 13.39

```
plot( NULL , xlab="treatment" , ylab="proportion pulled left" ,
    ylim=c(0,1) , xaxt="n" , xlim=c(1,4) )
axis( 1 , at=1:4 , labels=c("R/N","L/N","R/P","L/P") )
for ( i in 1:100 ) lines( 1:4 , p_raw_asim[i,] , col=grau(0.25) , lwd=2 )
```

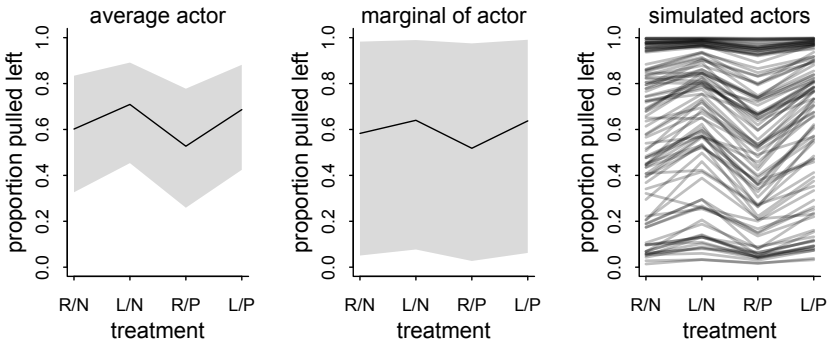

FIGURE 13.7. Posterior predictive distributions for the chimpanzees varying intercept model, `m13.4`. The solid lines are posterior means and the shaded regions are 80% percentile intervals. Left: Setting the varying intercept `a` to the mean `a_bar` produces predictions for an *average* actor. These predictions ignore uncertainty arising from variation among actors. Middle: Simulating varying intercepts using the posterior standard deviation among actors, `sigma_a`, produces predictions that account for variation among actors. Right: 100 simulated actors with unique intercepts sampled from the posterior. Each simulation maintains the same parameter values across all four treatments.

The result is shown in the right-hand plot of FIGURE 13.7. Each trend is a simulated actor, across all four treatments on the horizontal axis. It is much easier in this plot to see both the zig-zag impact of treatment and the variation among actors that is induced by the posterior distribution of `sigma_a`.

Also note the interaction of treatment and the variation among actors. Because this is a binomial model, in principle all parameters interact, due to ceiling and floor effects. For actors with very large intercepts, near the top of the plot, treatment has very little effect. These actors have strong handedness preferences. But actors with intercepts nearer the mean are influenced by treatment.

13.5.3. Post-stratification. A common problem is to use a non-representative sample of a population and to generate representative predictions for the same population. For example, we might survey potential voters, asking about their voting intentions. Such samples are nearly always biased—different groups respond to such surveys at different rates. So if we just use the survey average, we'll make the wrong prediction about the election. How can we do better?

One technique is **POST-STRATIFICATION**.[193] The idea is to fit a model in which each demographic slice of the population—a specific combination of age, economic, and educational variables for example—gets its own predicted voting intention. Then these predictions are re-weighted using general census information about the full voting population. Because there are usually many demographic categories, and samples can be small in some of them, post-stratification is often combined with multilevel modeling, in which case it is sometimes called MRP, pronounced "Mister P," for multilevel regression with post-stratification.

How does it work? Supposing you have estimates p_i for each demographic category i, then the post-stratified prediction for the whole population (not the sample) just re-weights these estimates using the number of individuals N_i in each category:

$$\frac{\sum_i N_i p_i}{\sum_i N_i}$$

Compute this for each sample in the posterior distribution, then you'll have a posterior distribution of predictions as usual.

Post-stratification does not always work. It is not justified when selection bias is itself caused by the outcome of interest. Suppose for example that responding to the survey R is influenced by age A, and that age A influences voting intention V: $R \leftarrow A \rightarrow V$. In that case it is possible to estimate the influence of A on V. But if $V \rightarrow R$, then there is little hope. Suppose for example that only supporters respond. Then $V = 1$ for everyone who responds. Selection on the outcome variable is one of the worst things that can happen in statistics.

13.6. Summary

This chapter has been an introduction to the motivation, implementation, and interpretation of basic multilevel models. It focused on varying intercepts, which achieve better estimates of baseline differences among clusters in the data. They achieve better estimates, because they simultaneously model the population of clusters and use inferences about the population to pool information among parameters. From another perspective, varying intercepts are adaptively regularized parameters, relying upon a prior that is itself learned from the data. All of this is a foundation for the next chapter, which extends these concepts to additional types of parameters and models.

13.7. Practice

Easy.

13E1. Which of the following priors will produce more *shrinkage* in the estimates? (a) $\alpha_{\text{TANK}} \sim \text{Normal}(0, 1)$; (b) $\alpha_{\text{TANK}} \sim \text{Normal}(0, 2)$.

13E2. Make the following model into a multilevel model.

$$\begin{aligned} y_i &\sim \text{Binomial}(1, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{GROUP}[i]} + \beta x_i \\ \alpha_{\text{GROUP}} &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \end{aligned}$$

13E3. Make the following model into a multilevel model.

$$\begin{aligned} y_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha_{\text{GROUP}[i]} + \beta x_i \\ \alpha_{\text{GROUP}} &\sim \text{Normal}(0, 10) \\ \beta &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{HalfCauchy}(0, 2) \end{aligned}$$

13E4. Write an example mathematical model formula for a Poisson regression with varying intercepts.

13E5. Write an example mathematical model formula for a Poisson regression with two different kinds of varying intercepts, a cross-classified model.

Medium.

13M1. Revisit the Reed frog survival data, `data(reedfrogs)`, and add the `predation` and `size` treatment variables to the varying intercepts model. Consider models with either main effect alone, both main effects, as well as a model including both and their interaction. Instead of focusing on inferences about these two predictor variables, focus on the inferred variation across tanks. Explain why it changes as it does across models.

13M2. Compare the models you fit just above, using WAIC. Can you reconcile the differences in WAIC with the posterior distributions of the models?

13M3. Re-estimate the basic Reed frog varying intercept model, but now using a Cauchy distribution in place of the Gaussian distribution for the varying intercepts. That is, fit this model:

$$\begin{aligned} s_i &\sim \text{Binomial}(n_i, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{TANK}[i]} \\ \alpha_{\text{TANK}} &\sim \text{Cauchy}(\alpha, \sigma) \\ \alpha &\sim \text{Normal}(0, 1) \\ \sigma &\sim \text{Exponential}(1) \end{aligned}$$

Compare the posterior means of the intercepts, α_{TANK}, to the posterior means produced in the chapter, using the customary Gaussian prior. Can you explain the pattern of differences?

13M4. Modify the cross-classified chimpanzees model `m13.4` so that the adaptive prior for blocks contains a parameter $\bar{\gamma}$ for its mean:

$$\begin{aligned} \gamma_j &\sim \text{Normal}(\bar{\gamma}, \sigma_\gamma) \\ \bar{\gamma} &\sim \text{Normal}(0, 1.5) \end{aligned}$$

Compare this model to `m13.4`. What has including $\bar{\gamma}$ done?

Hard.

13H1. In 1980, a typical Bengali woman could have 5 or more children in her lifetime. By the year 2000, a typical Bengali woman had only 2 or 3. You're going to look at a historical set of data, when contraception was widely available but many families chose not to use it. These data reside in `data(bangladesh)` and come from the 1988 Bangladesh Fertility Survey. Each row is one of 1934 women. There are six variables, but you can focus on three of them for this practice problem:

(1) `district`: ID number of administrative district each woman resided in
(2) `use.contraception`: An indicator (0/1) of whether the woman was using contraception
(3) `urban`: An indicator (0/1) of whether the woman lived in a city, as opposed to living in a rural area

The first thing to do is ensure that the cluster variable, `district`, is a contiguous set of integers. Recall that these values will be index values inside the model. If there are gaps, you'll have parameters for which there is no data to inform them. Worse, the model probably won't run. Look at the unique values of the `district` variable:

R code 13.40

```
sort(unique(d$district))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
[51] 51 52 53 55 56 57 58 59 60 61
```