12 Multilevel Models

In the year 1985, Clive Wearing lost his mind, but not his music.[154] 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 152) 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 14).

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 6. Then we'll finish by looking at contexts in which there is more than one type of clustering in the data. 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.[155] 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.

12.1. Example: Multilevel tadpoles

The heartwarming focus of this example are experiments exploring Reed frog (*Hyperolius spinigularis*) tadpole mortality.[156] 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 12.1

```
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**.[157] 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:

$$
\begin{aligned}
s_i &\sim \text{Binomial}(n_i, p_i) && \text{[likelihood]} \\
\text{logit}(p_i) &= \alpha_{\text{TANK}[i]} && \text{[unique log-odds for each tank } i\text{]} \\
\alpha_{\text{TANK}} &\sim \text{Normal}(0, 5) && \text{[weakly regularizing prior]}
\end{aligned}
$$

And you can fit this to the data in the standard way, using `map` or `map2stan`. We'll use `map2stan` from here onwards, because the next model will not work in `map`.

R code 12.2

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

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

# fit
m12.1 <- map2stan(
    alist(
        surv ~ dbinom( density , p ) ,
        logit(p) <- a_tank[tank] ,
        a_tank[tank] ~ dnorm( 0 , 5 )
    ),
    data=d )
```

If you inspect the estimates, `precis(m12.1,depth=2)`, you'll see 48 different intercept offsets, one for each tank. To get each tank's expected survival probability, just take one of the `a_tank` values and then use the logistic transform. So far there is nothing new here.

Now let's fit 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_tank` parameters a function of its own parameters. Here is the multilevel model, in mathematical form, with

the changes from the previous model highlighted in blue:

$$
\begin{aligned}
s_i &\sim \text{Binomial}(n_i, p_i) && \text{[likelihood]} \\
\text{logit}(p_i) &= \alpha_{\text{TANK}[i]} && \text{[log-odds for tank on row } i\text{]} \\
\alpha_{\text{TANK}} &\sim \text{Normal}(\alpha, \sigma) && \text{[varying intercepts prior]} \\
\alpha &\sim \text{Normal}(0, 1) && \text{[prior for average tank]} \\
\sigma &\sim \text{HalfCauchy}(0, 1) && \text{[prior for standard deviation of tanks]}
\end{aligned}
$$

Notice that the prior for the α_{TANK} intercepts is now a function of two parameters, α and σ. This is where the "multi" in multilevel arises.[158] The Gaussian distribution with mean α and standard deviation σ is the prior for each tank's intercept. But that prior itself has priors for α 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 α_{TANK}, and the prior is $\alpha_{\text{TANK}} \sim \text{Normal}(\alpha, \sigma)$. In the second level, the "outcome" variable is the vector of intercept parameters, α_{TANK}. The parameters are α and σ, and their priors are $\alpha \sim \text{Normal}(0, 1)$ and $\sigma \sim \text{HalfCauchy}(0, 1)$. For more explanation of the σ prior, see the Overthinking box on the next page.

These two parameters, α 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 9). There is no rule requiring the Gaussian distribution of varying effects, though. 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.

Fitting the model to data estimates 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 `map`. Why? Because the likelihood must now average over the level 2 parameters α and σ. But `map` 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 `map2stan`:

R code 12.3

```
m12.2 <- map2stan(
    alist(
        surv ~ dbinom( density , p ) ,
        logit(p) <- a_tank[tank] ,
        a_tank[tank] ~ dnorm( a , sigma ) ,
        a ~ dnorm(0,1) ,
        sigma ~ dcauchy(0,1)
    ), data=d , iter=4000 , chains=4 )
```

This model fit provides estimates for 50 parameters: one overall sample intercept α, the variance 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, `m12.1`, with the new multilevel model:

R code 12.4

```
compare( m12.1 , m12.2 )
```

```
        WAIC pWAIC dWAIC weight    SE  dSE
m12.2 1010.2  38.0   0.0      1 37.94   NA
m12.1 1023.3  49.4  13.1      0 43.01 6.54
```

There are two facts to note here. First, the multilevel model has only 38 effective parameters. There are 12 fewer effective parameters than actual parameters, because the prior assigned to each intercept shrinks them all towards the mean α. In this case, the prior is reasonably strong. Check the mean of `sigma` with `precis` or `coef` 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.[159] Second, notice that the multilevel model `m12.2` has fewer effective parameters than the ordinary fixed model `m12.1`. 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: MAP fails, MCMC wins. Why doesn't MAP estimation, using for example `map`, 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 MAP estimation 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. More generally, as models become more complex, a phenomenon known as *concentration of measure* guarantees that the posterior mode will be far from the posterior median. So we really need to give up optimization as a strategy. One robust solution is MCMC.

To appreciate the impact of this adaptive regularization, let's plot and compare the posterior medians from models `m12.1` and `m12.2`. 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 medians.

R code 12.5

```
# extract Stan samples
post <- extract.samples(m12.2)

# compute median intercept for each tank
# also transform to probability with logistic
d$propsurv.est <- logistic( apply( post$a_tank , 2 , median ) )
```

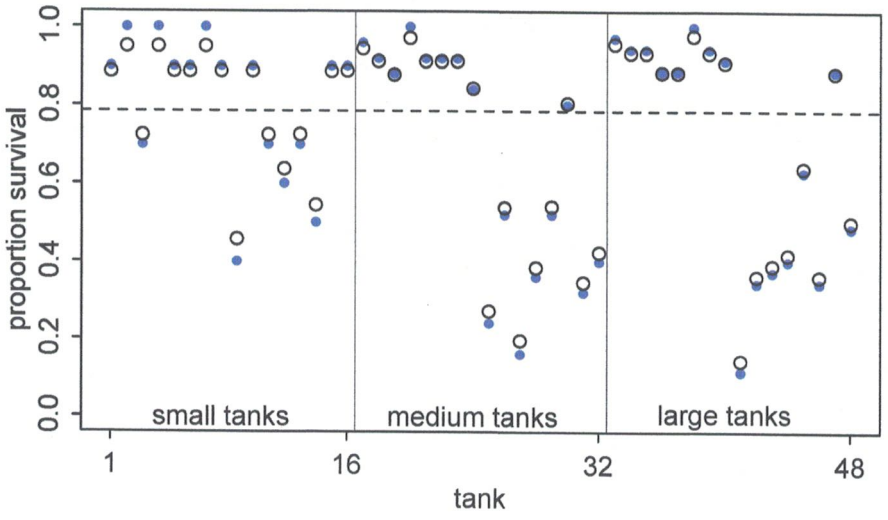

FIGURE 12.1. Empirical proportions of survivors in each tadpole tank, shown by the filled blue points, plotted with the 48 per-tank estimates from the multilevel model, shown by the black circles. The dashed line locates the overall average proportion of survivors across all tanks. The vertical lines divide tanks with different initial densities of tadpoles: small tanks (10 tadpoles), medium tanks (25), and large tanks (35). In every tank, the posterior median 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.

```
# 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 medians
points( d$propsurv.est )

# mark posterior median probability across tanks
abline( h=logistic(median(post$a)) , 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 12.1. 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 6). 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 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 12.6

```
# 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$a[i],post$sigma[i]) , add=TRUE ,
    col=col.alpha("black",0.2) )

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

# transform to probability and visualize
```

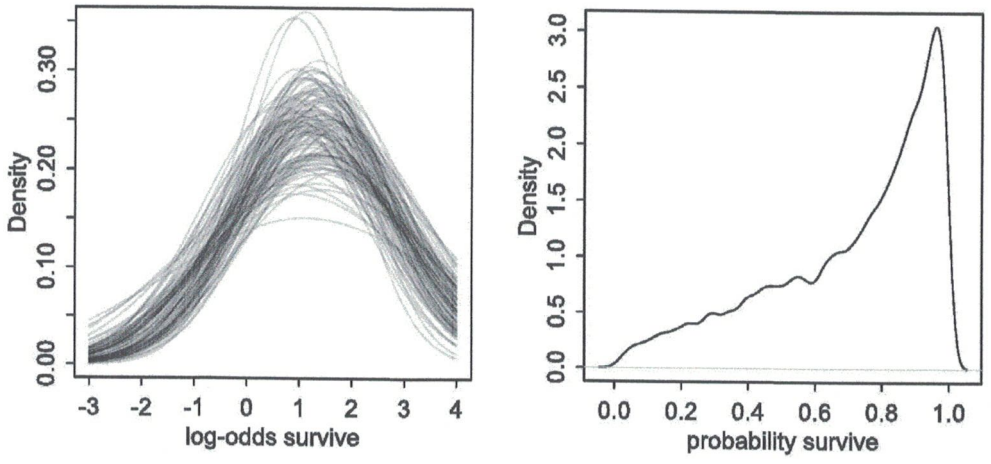

FIGURE 12.2. The inferred population of survival across tanks. Left: 100 Gaussian distributions of the log-odds of survival, sampled from the posterior of `m12.2`. Right: Survival probabilities for 8000 new simulated tanks, averaging over the posterior distribution on the left.

```
dens( logistic(sim_tanks) , xlab="probability survive" )
```

The results are displayed in FIGURE 12.2. 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 346), 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 half-Cauchy priors for variance components, the σ parameters that estimate the variation across clusters in the data. These Cauchy priors work very well in routine multilevel modeling. 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. 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 Cauchy prior has a very thick and long tail, extending into very large values. 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 iterations.

To improve such a model, instead of using half-Cauchy priors for the variance components, you can use exponential priors. For example:

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

The exponential prior—`dexp(1)` in R code—has a much thinner tail than the Cauchy does. This induces more conservatism in estimates and can help your Markov chain converge correctly. The exponential is also the maximum entropy prior for the standard deviation, provided all we want to say *a priori* is the expected value. That is to say that the only information contained in an exponential prior is the mean value and the positive constraint.

Again, using the exponential instead of the Cauchy isn't usually necessary. But there are cases, especially with non-linear models with ceiling or floor effects, in which the variance components can be only weakly identified. In those cases, you are going to have to add more strongly regularizing priors in order to make any inference at all. And of course, it is typically useful to try different priors to ensure that inference either is insensitive to them or rather to measure how inference is altered.

12.2. Varying effects and the underfitting/overfitting trade-off

Varying intercepts are just regularized estimates, but adaptively regularized by estimating how diverse the clusters are while estimating the features of each cluster. This fact is not easy to grasp, 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 6.

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.[160] 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 12.1 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.

12.2.1. 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":

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

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

- α, 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_{POND}, 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.

12.2.2. 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 12.7

```
a <- 1.4
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 `map2stan`, works. See the Overthinking box at the bottom of the page for an explanation.

The values $\alpha = 1.4$ 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 α and standard deviation σ:

R code 12.8

```
a_pond <- rnorm( nponds , mean=a , 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 12.9

```
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 12.10

```
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 `map2stan` 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 `map2stan` will resolve the issue.

12.2.3. 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:

```
dsim$si <- rbinom( nponds , prob=logistic(dsim$true_a) , size=dsim$ni )
```

R code 12.11

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`.

12.2.4. 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.

```
dsim$p_nopool <- dsim$si / dsim$ni
```

R code 12.12

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.

12.2.5. 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.

```
m12.3 <- map2stan(
    alist(
        si ~ dbinom( ni , p ),
        logit(p) <- a_pond[pond],
        a_pond[pond] ~ dnorm( a , sigma ),
        a ~ dnorm(0,1),
        sigma ~ dcauchy(0,1)
    ),
```

R code 12.13

```
    data=dsim , iter=1e4 , warmup=1000 )
```

We've fit the basic varying intercept model above. You can take a look at the estimates for α and σ with the usual `precis` approach:

R code 12.14
```
precis(m12.3,depth=2)
```

```
             Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
a_pond[1]    1.45   0.95      -0.11       2.89  9000    1
a_pond[2]    1.47   0.95      -0.02       2.96  9000    1
...
a_pond[59]   1.81   0.47       1.02       2.52  7314    1
a_pond[60]   2.03   0.50       1.24       2.82  9000    1
a            1.13   0.23       0.78       1.50  5848    1
sigma        1.59   0.22       1.25       1.93  2705    1
```

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

Now that we've found these estimates, 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 12.15
```
estimated.a_pond <- as.numeric( coef(m12.3)[1:60] )
dsim$p_partpool <- logistic( estimated.a_pond )
```

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 12.16
```
dsim$p_true <- logistic( dsim$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 12.17
```
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 12.18
```
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 12.3. Your own plot will look different, because of simulation variance. The pattern displayed in the figure is the central tendency. To see how to quickly re-run the model on newly simulated data, without re-compiling, see the Overthinking box at the end of this section.

The filled blue points in FIGURE 12.3 display the no-pooling estimates. The black circles show the varying effect estimates. The horizontal axis is the pond index, from 1 through

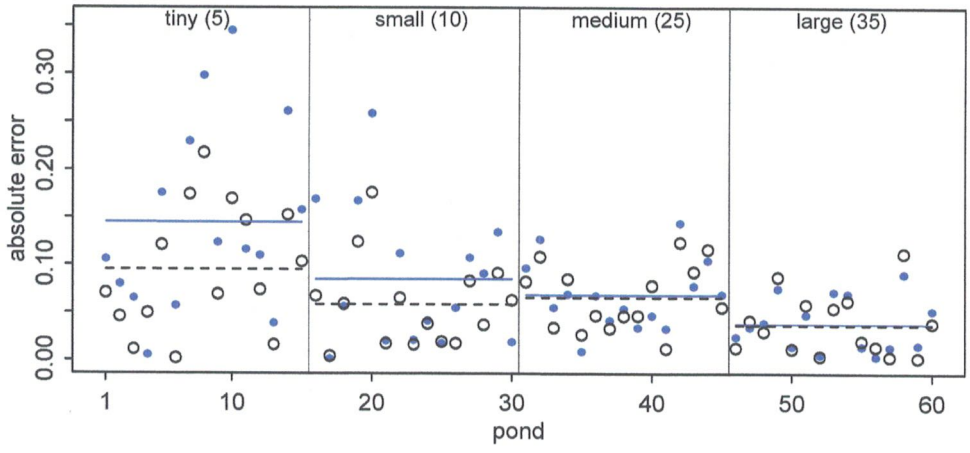

FIGURE 12.3. Error of no-pooling and partial pooling estimates, for the simulated tadpole ponds. The horizontal axis displays pond number. The vertical axis measures the absolute error in the predicted proportion of survivors, compared to the true value used in the simulation. The higher the point, the 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.

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.

The first thing to notice about this 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, usually. In the small ponds, sample size is small, and neither kind of estimate can work magic. Therefore, prediction suffers on the left side of the plot. Second, note that the blue line is always above 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. So 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.

Okay, so what are we to make of all of this? Remember, back in FIGURE 12.1 (page 361), 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 6, 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 `m12.3` 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 12.19

```
a <- 1.4
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$si <- rbinom( nponds,prob=logistic( dsim$true_a ),size=dsim$ni )
dsim$p_nopool <- dsim$si / dsim$ni
newdat <- list(si=dsim$si,ni=dsim$ni,pond=1:nponds)
m12.3new <- map2stan( m12.3 , data=newdat , iter=1e4 , warmup=1000 )
```

The `map2stan` function reuses the compiled model in `m12.3`, passes it the new data, and returns the new samples in `m12.3new`. This is a useful trick, in case you want to perform a simulation study of a particular model structure. And if you ever want to extract the actual compiled Stan model, it is held in `m12.3@stanfit`, and you can always view its code with `stancode(m12.3)` and the input data (which is augmented a bit) with `m12.3@data`.

12.3. More than one type of cluster

We can use and often should use more than one type of cluster in the same model. For example, the observations in `data(chimpanzees)`, which you met back in Chapter 10, 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 13). 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.

12.3.1. Multilevel chimpanzees. Let's proceed by taking the full chimpanzees model from Chapter 10 (`m10.4`, page 299) and first adding varying intercepts on actor. To add varying intercepts to this model, we just replace the fixed regularizing prior with an adaptive prior. But this time, I'll put the mean α up in the linear model, rather than down in the prior. Why? Because it will pave the way to adding more varying effects later. You'll see why, once we've pushed forward a little.

Here is the multilevel chimpanzees model in mathematical form, with the varying intercept components highlighted in blue:

$$\begin{aligned} L_i &\sim \text{Binomial}(1, p_i) \\ \text{logit}(p_i) &= \alpha + \alpha_{\text{ACTOR}[i]} + (\beta_P + \beta_{PC} C_i) P_i \\ \alpha_{\text{ACTOR}} &\sim \text{Normal}(0, \sigma_{\text{ACTOR}}) \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta_P &\sim \text{Normal}(0, 10) \\ \beta_{PC} &\sim \text{Normal}(0, 10) \\ \sigma_{\text{ACTOR}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

Notice that α is inside the linear model, not inside the Gaussian prior for α_{ACTOR}. This is mathematically equivalent to what you did with the tadpoles earlier in the chapter. You can always take the mean out of a Gaussian distribution and treat the distribution as a constant plus a Gaussian distribution centered on zero.

This might seem a little weird at first, so it might help train your intuition by experimenting in R. These two lines of code sample values from two identical Gaussian distributions, with mean 10 and standard deviation 1:

R code 12.20

```
y1 <- rnorm( 1e4 , 10 , 1 )
y2 <- 10 + rnorm( 1e4 , 0 , 1 )
```

Inspect the distributions of values in `y1` and `y2`. You'll see they are the same. This feature of the Gaussian distribution arises from the independence of the mean and standard deviation. Most distributions do not have this property. But we'll exploit it here. And sometimes a given combination of model and data is more efficiently fit using one form or the other. I'll say more about this in the next chapter.

Here's the corresponding `map2stan` code for the model with varying intercepts on `actor`, but not yet on `block`. Note that the linear model contains α, the varying intercepts mean. The adaptive prior for the intercepts themselves has a mean of zero.

R code 12.21

```
library(rethinking)
data(chimpanzees)
d <- chimpanzees
d$recipient <- NULL     # get rid of NAs

m12.4 <- map2stan(
    alist(
        pulled_left ~ dbinom( 1 , p ) ,
        logit(p) <- a + a_actor[actor] + (bp + bpC*condition)*prosoc_left ,
        a_actor[actor] ~ dnorm( 0 , sigma_actor ),
        a ~ dnorm(0,10),
        bp ~ dnorm(0,10),
        bpC ~ dnorm(0,10),
        sigma_actor ~ dcauchy(0,1)
    ) ,
    data=d , warmup=1000 , iter=5000 , chains=4 , cores=3 )
```

Inspect the trace plot, `plot(m12.4)`, and the posterior distribution for `sigma_actor`. Make sure the effective numbers of samples and `Rhat` values look alright. If you need to review these MCMC diagnostics, glance back at Chapter 8.

Now that the mean of the population of actors, α (`a`), is in the linear model, it's important to notice now that the `a_actor` parameters are *deviations* from `a`. So for any given row i, the total intercept is $\alpha + \alpha_{\text{ACTOR}[i]}$. The part that varies across actors is just the deviation from the grand mean α. To compute the total intercept for each `actor`, you need to add samples of `a` to samples of `a_actor`:

R code 12.22

```
post <- extract.samples(m12.4)
total_a_actor <- sapply( 1:7 , function(actor) post$a + post$a_actor[,actor] )
round( apply(total_a_actor,2,mean) , 2 )
```

```
[1] -0.71  4.59 -1.02 -1.02 -0.71  0.23  1.76
```

12.3.2. Two types of cluster. 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:

$$
\begin{aligned}
L_i &\sim \text{Binomial}(1, p_i) \\
\text{logit}(p_i) &= \alpha + \alpha_{\text{ACTOR}[i]} + \alpha_{\text{BLOCK}[i]} + (\beta_P + \beta_{PC} C_i) P_i \\
\alpha_{\text{ACTOR}} &\sim \text{Normal}(0, \sigma_{\text{ACTOR}}) \\
\alpha_{\text{BLOCK}} &\sim \text{Normal}(0, \sigma_{\text{BLOCK}}) \\
\alpha &\sim \text{Normal}(0, 10) \\
\beta_P &\sim \text{Normal}(0, 10) \\
\beta_{PC} &\sim \text{Normal}(0, 10) \\
\sigma_{\text{ACTOR}} &\sim \text{HalfCauchy}(0, 1) \\
\sigma_{\text{BLOCK}} &\sim \text{HalfCauchy}(0, 1)
\end{aligned}
$$

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_{ACTOR} and σ_{BLOCK}, respectively. Finally, note that there is only one global mean parameter α, and both of the varying intercept parameters are centered at zero. We can't identify a separate mean for each varying intercept type, because both intercepts are added to the same linear prediction. So it is conventional to define varying intercepts with a mean of zero, so there's no risk of accidentally creating hard-to-identify parameters. There's a practice problem at the end of the chapter that leads you to explore what happens when you forget and instead include two grand mean α parameters.

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

R code 12.23

```
# prep data
d$block_id <- d$block  # name 'block' is reserved by Stan

m12.5 <- map2stan(
    alist(
        pulled_left ~ dbinom( 1 , p ),
        logit(p) <- a + a_actor[actor] + a_block[block_id] +
                    (bp + bpc*condition)*prosoc_left,
        a_actor[actor] ~ dnorm( 0 , sigma_actor ),
        a_block[block_id] ~ dnorm( 0 , sigma_block ),
        c(a,bp,bpc) ~ dnorm(0,10),
        sigma_actor ~ dcauchy(0,1),
        sigma_block ~ dcauchy(0,1)
    ) ,
    data=d, warmup=1000 , iter=6000 , chains=4 , cores=3 )
```

If all goes well, you'll end up with 20,000 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 might see for the first time a warning about *divergent iterations*:

```
Warning message:
In map2stan(alist(pulled_left ~ dbinom(1, p), logit(p) <- a + a_actor[actor] +  :
  There were 3 divergent iterations during sampling.
Check the chains (trace plots, n_eff, Rhat) carefully to ensure they are valid.
```

We'll have a lot more to say about these in the next chapter. For now, they are safe to ignore. Just do as stated and inspect `n_eff` and `Rhat`.

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

R code 12.24

```
precis(m12.5,depth=2) # depth=2 displays varying effects
plot(precis(m12.5,depth=2)) # also plot
```

```
              Mean StdDev lower 0.89 upper 0.89 n_eff Rhat
a_actor[1]   -1.17   0.93      -2.57       0.29  2333    1
a_actor[2]    4.14   1.59       1.92       6.33  4543    1
a_actor[3]   -1.48   0.94      -2.91      -0.02  2310    1
a_actor[4]   -1.48   0.94      -2.92      -0.01  2360    1
a_actor[5]   -1.17   0.94      -2.63       0.27  2354    1
a_actor[6]   -0.22   0.93      -1.65       1.25  2359    1
a_actor[7]    1.32   0.96      -0.15       2.84  2496    1
a_block[1]   -0.18   0.22      -0.53       0.11  3848    1
a_block[2]    0.04   0.18      -0.23       0.34  8595    1
a_block[3]    0.05   0.19      -0.23       0.35  7237    1
a_block[4]    0.00   0.18      -0.30       0.28  9532    1
a_block[5]   -0.04   0.18      -0.34       0.25  8327    1
a_block[6]    0.11   0.20      -0.17       0.43  5420    1
a             0.46   0.92      -0.98       1.88  2263    1
bp            0.82   0.26       0.41       1.25  6890    1
bpc          -0.13   0.30      -0.62       0.33  8360    1
sigma_actor   2.25   0.90       1.02       3.33  4892    1
sigma_block   0.22   0.18       0.01       0.43  2079    1
```

The `precis` plot is shown in the left-hand part of FIGURE 12.4.

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 the most common reason is that some parameter spends a lot of time near a boundary. Here, that parameter is `sigma_block`. It spends a lot of time near its minimum of zero. As a consequence, you may also see a warning about "divergent iterations." You can wait until the next chapter to explore what these mean and what to do about them. For now, you can trust the `Rhat` values above, but later you'll see how to make sampling more efficient for models like these.

Second, compare `sigma_actor` to `sigma_block` 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:

R code 12.25

```
post <- extract.samples(m12.5)
dens( post$sigma_block , xlab="sigma" , xlim=c(0,4) )
dens( post$sigma_actor , col=rangi2 , lwd=2 , add=TRUE )
text( 2 , 0.85 , "actor" , col=rangi2 )
text( 0.75 , 2 , "block" )
```

And this plot appears on the right in FIGURE 12.4. While there's uncertainty about the variation among actors, this model is confident that actors vary more than blocks. You can easily

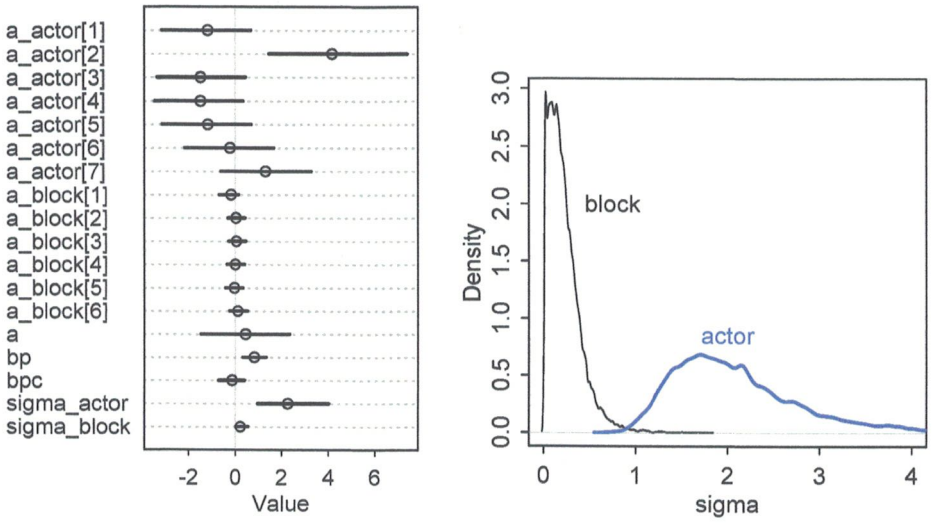

FIGURE 12.4. Left: Posterior means and 89% highest density intervals for `m12.5`. The greater variation across actors than blocks can be seen immediately in the `a_actor` and `a_block` distributions. Right: Posterior distributions of the standard deviations of varying intercepts by actor (blue) and experimental block (black).

see this variation in the varying intercept estimates: the `a_actor` distributions are much more scattered than are the `a_block` distributions.

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:

R code 12.26

```
compare(m12.4,m12.5)
```

```
       WAIC pWAIC dWAIC weight    SE  dSE
m12.4 531.5   8.1   0.0   0.65 19.50   NA
m12.5 532.7  10.5   1.2   0.35 19.74 1.94
```

Look at the `pWAIC` column, which reports the "effective number of parameters." While `m12.5` has 7 more parameters than `m12.4` does, it has only about 2.5 more effective parameters. Why? Because the posterior distribution for `sigma_block` ended up close to zero. This means each of the 6 `a_block` parameters is strongly shrunk towards zero—they are relatively inflexible. In contrast, the `a_actor` 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_actor` parameters contributes much more to the `pWAIC` value.

You might also notice that the difference in WAIC between these models is small, only 1.2. This is especially small compared the standard deviation of the difference, 1.94. 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" `m12.4` 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 `a_block` and `sigma_block` estimates tell us why. Furthermore, the standard error of the difference in WAIC between the models is twice as large as the difference itself. By retaining and reporting both models, we and our readers learn more about the experiment.

12.3.3. Even more clusters. Adding more types of clusters proceeds the same way. At some point the model may become too complex to reliably fit to data. But Hamiltonian Monte Carlo is very capable with varying effects. It can easily handle tens of thousands of varying effect parameters. Sampling will be slow in such cases, but it will work.

So don't be shy—if you have a good theoretical reason to include a cluster variable, then you also have good theoretical reason to partially pool its parameters. As you've seen, the overfitting risk induced by including varying intercepts can be quite small, when there is little variation among the clusters. The multilevel model adaptively regularizes, helping us discover the relevance of different kinds of clusters within the data. In this way, you can think of the `sigma` parameter for each cluster as a crude measure of the cluster's relevance for explaining variation in the outcome.

12.4. 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.

Another role for constructing implied predictions is in computing **INFORMATION CRITERIA**, like DIC 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 6.

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 that reflects shrinkage.

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.

12.4.1. 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. But otherwise, there are no tricks.

For example, in `data(chimpanzees)`, there are 7 unique actors. These are the clusters. The varying intercepts model, `m12.4`, 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 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 10. Here it is again, computing and plotting posterior predictions for actor number 2:

R code 12.27

```
chimp <- 2
d.pred <- list(
    prosoc_left = c(0,1,0,1),    # right/left/right/left
    condition = c(0,0,1,1),      # control/control/partner/partner
    actor = rep(chimp,4)
)
link.m12.4 <- link( m12.4 , data=d.pred )
pred.p <- apply( link.m12.4 , 2 , mean )
pred.p.PI <- apply( link.m12.4 , 2 , PI )
```

And the plotting code is exactly the same as before (page 297).

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 12.28

```
post <- extract.samples(m12.4)
str(post)
```

```
List of 5
 $ a_actor    : num [1:8000, 1:7] -1.842 -0.225 -1.811 -0.759 -1.882 ...
 $ a          : num [1:8000(1d)] 1.291 -0.632 0.285 -0.109 1.229 ...
 $ bp         : num [1:8000(1d)] 1 1.064 1.087 0.254 0.908 ...
 $ bpC        : num [1:8000(1d)] -0.272 -0.539 -0.295 0.375 -0.218 ...
 $ sigma_actor: num [1:8000(1d)] 2.13 2.49 2.32 1.51 4.12 ...
```

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

R code 12.29

```
dens( post$a_actor[,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 12.30

```
p.link <- function( prosoc_left , condition , actor ) {
    logodds <- with( post ,
        a + a_actor[,actor] + (bp + bpC * condition) * prosoc_left
    )
    return( logistic(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_actor`. Now to compute predictions:

R code 12.31

```
prosoc_left <- c(0,1,0,1)
condition <- c(0,0,1,1)
pred.raw <- sapply( 1:4 , function(i) p.link(prosoc_left[i],condition[i],2) )
pred.p <- apply( pred.raw , 2 , mean )
pred.p.PI <- apply( pred.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.

12.4.2. Posterior prediction for new clusters. Often, the particular clusters in the sample are not of any enduring interest. In the chimpanzees data, for example, these particular 7 chimpanzees are just seven individuals. We'd like to make inferences about the whole species, not just those seven individuals. So the individual actor intercepts aren't of interest, but the distribution of them definitely is.

One way to grasp the task of construction 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_actor` parameter estimates, because those apply to other individuals. But we can make good use of the `a` and `sigma_actor` parameters, because those describe the population of actors.

First, let's see how to construct posterior predictions for a now, previously unobserved *average* actor. By "average," I mean an individual chimpanzee with an intercept exactly at `a` (α), the population mean. This simultaneously implies a varying intercept of zero. 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.

The first step is to make a new data list to compute predictions over. You've done this in previous chapters. Here is our new list, representing the four different treatments:

R code 12.32

```
d.pred <- list(
    prosoc_left = c(0,1,0,1),    # right/left/right/left
    condition = c(0,0,1,1),      # control/control/partner/partner
    actor = rep(2,4) )           # placeholder
```

Next, we're going to make a matrix of zeros, to replace the varying intercept samples. It's easiest to just keep the same dimension as the original matrix. In this case that means using 1000 samples for each of 7 actors. But all of the samples will be set to zero:

R code 12.33

```
# replace varying intercept samples with zeros
# 1000 samples by 7 actors
a_actor_zeros <- matrix(0,1000,7)
```

That's the only new trick. Now we just pass this new matrix to `link` using the optional `replace` argument. Make sure the new matrix is named the same as the varying intercept matrix, `a_actor`. Otherwise it won't replace anything that appears in the model.

R code 12.34

```
# fire up link
# note use of replace list
link.m12.4 <- link( m12.4 , n=1000 , data=d.pred ,
    replace=list(a_actor=a_actor_zeros) )

# summarize and plot
pred.p.mean <- apply( link.m12.4 , 2 , mean )
pred.p.PI <- apply( link.m12.4 , 2 , PI , prob=0.8 )
plot( 0 , 0 , type="n" , xlab="prosoc_left/condition" ,
    ylab="proportion pulled left" , ylim=c(0,1) , xaxt="n" ,
    xlim=c(1,4) )
axis( 1 , at=1:4 , labels=c("0/0","1/0","0/1","1/1") )
lines( 1:4 , pred.p.mean )
shade( pred.p.PI , 1:4 )
```

The result is displayed in FIGURE 12.5, on the left. The gray region shows the 80% 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_actor` in the calculation. We can again smuggle this into `link` by using the `replace` argument. This time however, we'll simulate a matrix of new varying intercepts from a Gaussian distribution defined by the adaptive prior in the model itself:

$$\alpha_{\text{ACTOR}} \sim \text{Normal}(0, \sigma_{\text{ACTOR}})$$

This implies that once we have samples for σ_{ACTOR}, we can simulate new actor intercepts from this distribution. Here's the code to do just that, using `rnorm`:

R code 12.35

```
# replace varying intercept samples with simulations
post <- extract.samples(m12.4)
a_actor_sims <- rnorm(7000,0,post$sigma_actor)
a_actor_sims <- matrix(a_actor_sims,1000,7)
```

Now pass the simulated intercepts into `link`. Note the `replace` list, which inserts the simulations into the posterior.

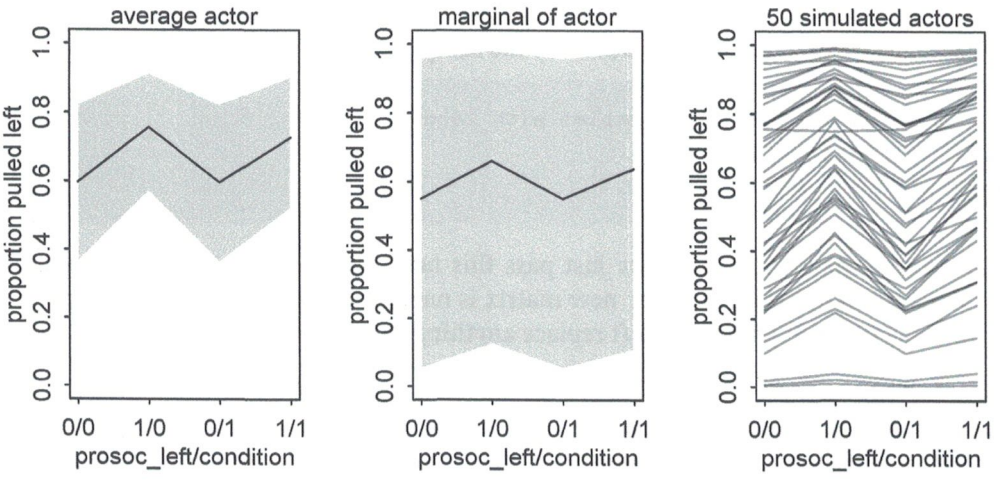

FIGURE 12.5. Posterior predictive distributions for the chimpanzees varying intercept model, `m12.4`. The solid lines are posterior means and the shaded regions are 80% percentile intervals. Left: Setting the varying intercept `a_actor` to zero produces predictions for an *average* actor. These predictions ignore uncertainty arising from variation among actors. Middle: Simulating varying intercepts using the posterior standard deviation among actors, `sigma_actor`, produces predictions that account for variation among actors. Right: 50 simulated actors with unique intercepts sampled from the posterior. Each simulation maintains the same parameter values across all four treatments.

R code 12.36

```
link.m12.4 <- link( m12.4 , n=1000 , data=d.pred ,
    replace=list(a_actor=a_actor_sims) )
```

Summarizing and plotting is exactly as before, and the result is displayed in the middle of FIGURE 12.5. 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.

What we'll do now is write a new function that simulates a new actor from the estimated population of actors and then computes probabilities of pulling the left lever for each of the four treatments. These simulations will not average over uncertainty in the posterior. We'll get that uncertainty into the plot by using multiple simulations, each with a different sample from the posterior. Here's the function:

R code 12.37

```
post <- extract.samples(m12.4)
sim.actor <- function(i) {
    sim_a_actor <- rnorm( 1 , 0 , post$sigma_actor[i] )
    P <- c(0,1,0,1)
    C <- c(0,0,1,1)
    p <- logistic(
        post$a[i] +
        sim_a_actor +
        (post$bp[i] + post$bpC[i]*C)*P
    )
    return(p)
}
```

This function takes a single argument, `i`, which is just the index of a sample from the posterior distribution. It then draws a random intercept for the actor, using `rnorm` and a particular value of `sigma_actor`. Then it computes probabilities `p` for each of the four treatments, using the same linear model, but with different predictor values inside the `P` and `C` vectors. Because these vectors are of length 4, the code spits out 4 values for `p`.

Now to use this function to plot 50 simulations:

R code 12.38

```
# empty plot
plot( 0 , 0 , type="n" , xlab="prosoc_left/condition" ,
    ylab="proportion pulled left" , ylim=c(0,1) , xaxt="n" , xlim=c(1,4) )
axis( 1 , at=1:4 , labels=c("0/0","1/0","0/1","1/1") )

# plot 50 simulated actors
for ( i in 1:50 ) lines( 1:4 , sim.actor(i) , col=col.alpha("black",0.5) )
```

The result is shown in the right-hand plot of FIGURE 12.5. 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_actor`.

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.

12.4.3. Focus and multilevel prediction. All of this is confusing at first. There is no uniquely correct way to always construct the predictions, and the calculations themselves probably seem a little magical. In time, it makes a lot more sense. The fact is that multilevel models contain parameters with different **FOCUS**. Focus here means which level of the model the

parameter makes direct predictions for. It helps to organize the issue into three common cases.

First, when retrodicting the sample, the parameters that describe the population of clusters, such as α and σ_{ACTOR} in `m12.4`, do not influence prediction directly. Recall that these population parameters are often called **HYPERPARAMETERS**, as they are parameters for parameters. These hyperparameters had their effects during estimation, by shrinking the varying effect parameters towards a common mean. The prediction focus here is on the top level of parameters, not the deeper hyperparameters.

Second, the same is true when forecasting a new observation for a cluster that was present in the sample. For example, if we want to predict what chimpanzee number 2 will do in the next experiment, we should probably bet she'll pull the left lever, because her varying intercept was very large. The focus is again on the top level.

The third case is different. When instead we wish to forecast for some new cluster that was not present in the sample, such as a new individual or school or year or location, then we need the hyper-parameters. The hyper-parameters tell us how to forecast a new cluster, by generating a distribution of new per-cluster intercepts. This is what we did in the previous section, simulating new chimpanzees.

This is also the right thing to do whenever varying effects are used to model **OVER-DISPERSION** (page 363). In that case, we need to simulate intercepts in order to account for the over-dispersion. Here's a quick example, using the Oceanic societies example from Chapter 10, but now adding a varying intercept to each society. Here's the mathematical form of the model, with the varying intercept pieces highlighted in blue:

$$\begin{aligned} T_i &\sim \text{Poisson}(\mu_i) \\ \log(\mu_i) &= \alpha + \alpha_{\text{SOCIETY}[i]} + \beta_P \log P_i \\ \alpha &\sim \text{Normal}(0, 10) \\ \beta_P &\sim \text{Normal}(0, 1) \\ \alpha_{\text{SOCIETY}} &\sim \text{Normal}(0, \sigma_{\text{SOCIETY}}) \\ \sigma_{\text{SOCIETY}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

T is `total_tools`, P is `population`, and i indexes each society. The above is just a varying intercept model, but with a varying intercept for every observation. As a result, σ_{SOCIETY} ends up being an estimate of the over-dispersion among societies. Another way to think of this is that the varying intercepts α_{SOCIETY} are residuals for each society. By also estimating the distribution of these residuals, we get an estimate of the excess variation, relative to the Poisson expectation.

And here is the code to fit the over-dispersed Poisson model:

R code 12.39

```
# prep data
library(rethinking)
data(Kline)
d <- Kline
d$logpop <- log(d$population)
d$society <- 1:10

# fit model
m12.6 <- map2stan(
```

```
    alist(
        total_tools ~ dpois(mu),
        log(mu) <- a + a_society[society] + bp*logpop,
        a ~ dnorm(0,10),
        bp ~ dnorm(0,1),
        a_society[society] ~ dnorm(0,sigma_society),
        sigma_society ~ dcauchy(0,1)
    ),
    data=d ,
    iter=4000 , chains=3 )
```

This model samples very efficiently, despite using 13 parameters to describe 10 observations. Remember: Varying effect parameters are adaptively regularized. So they are not completely flexible and induce much less overfitting risk. In this case, WAIC should tell you that the effective number of parameters is about 5, not 13. If you have hundreds or thousands of observations in the data, this approach still works fine. You just end up with hundreds or thousands of varying intercept estimates. You won't care about the estimates themselves. But you will care about the hyperparameters that describe the population of varying intercepts.

Now to generate posterior predictions that visualize the over-dispersion. You can display posterior predictions (retrodictions) by using `postcheck(m12.6)`. But those predictions just use the varying intercepts, `a_society`, directly. They do not use the hyper-parameters. To instead see the general trend that the model expects, we'll need to simulate counterfactual societies, using the hyper-parameters α and σ_{SOCIETY}. This is the same procedure that we used for new chimpanzee actors earlier.

R code 12.40

```
post <- extract.samples(m12.6)
d.pred <- list(
    logpop = seq(from=6,to=14,length.out=30),
    society = rep(1,30)
)
a_society_sims <- rnorm(20000,0,post$sigma_society)
a_society_sims <- matrix(a_society_sims,2000,10)
link.m12.6 <- link( m12.6 , n=2000 , data=d.pred ,
    replace=list(a_society=a_society_sims) )
```

And this code will display the raw data and the new prediction envelope:

R code 12.41

```
# plot raw data
plot( d$logpop , d$total_tools , col=rangi2 , pch=16 ,
    xlab="log population" , ylab="total tools" )

# plot posterior median
mu.median <- apply( link.m12.6 , 2 , median )
lines( d.pred$logpop , mu.median )

# plot 97%, 89%, and 67% intervals (all prime numbers)
mu.PI <- apply( link.m12.6 , 2 , PI , prob=0.97 )
shade( mu.PI , d.pred$logpop )
```

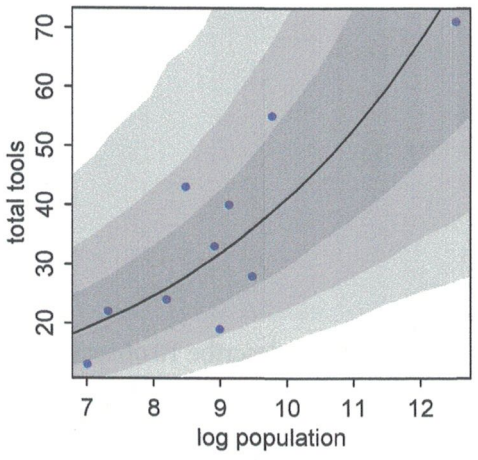

FIGURE 12.6. Posterior predictions for the over-dispersed Poisson island model, `m12.6`. The shaded regions are, inside to out: 67%, 89%, and 97% intervals of the expected mean. Marginalizing over the varying intercepts results in a much wider prediction region than we'd expect under a pure Poisson process.

```
mu.PI <- apply( link.m12.6 , 2 , PI , prob=0.89 )
shade( mu.PI , d.pred$logpop )
mu.PI <- apply( link.m12.6 , 2 , PI , prob=0.67 )
shade( mu.PI , d.pred$logpop )
```

The result is displayed in FIGURE 12.6. The envelope of predictions is a lot wider here than it was back in Chapter 10. This is a consequence of the varying intercepts, combined with the fact that there is much more variation in the data than a pure-Poisson model anticipates.

12.5. 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.

12.6. Practice

Easy.

12E1. Which of the following priors will produce more *shrinkage* in the estimates? (a) $\alpha_{\text{TANK}} \sim \text{Normal}(0, 1)$; (b) $\alpha_{\text{TANK}} \sim \text{Normal}(0, 2)$.

12E2. Make the following model into a multilevel model.

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

12E3. Make the following model into a multilevel model.

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

12E4. Write an example mathematical model formula for a Poisson regression with varying intercepts.

12E5. Write an example mathematical model formula for a Poisson regression with two different kinds of varying intercepts, a cross-classified model.

Medium.

12M1. 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.

12M2. Compare the models you fit just above, using WAIC. Can you reconcile the differences in WAIC with the posterior distributions of the models?

12M3. 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:

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

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?

12M4. Fit the following cross-classified multilevel model to the `chimpanzees` data:

$$\begin{aligned} L_i &\sim \text{Binomial}(1, p_i) \\ \text{logit}(p_i) &= \alpha_{\text{ACTOR}[i]} + \alpha_{\text{BLOCK}[i]} + (\beta_P + \beta_{PC} C_i) P_i \\ \alpha_{\text{ACTOR}} &\sim \text{Normal}(\alpha, \sigma_{\text{ACTOR}}) \\ \alpha_{\text{BLOCK}} &\sim \text{Normal}(\gamma, \sigma_{\text{BLOCK}}) \\ \alpha, \gamma, \beta_P, \beta_{PC} &\sim \text{Normal}(0, 10) \\ \sigma_{\text{ACTOR}}, \sigma_{\text{BLOCK}} &\sim \text{HalfCauchy}(0, 1) \end{aligned}$$

Each of the parameters in those comma-separated lists gets the same independent prior. Compare the posterior distribution to that produced by the similar cross-classified model from the chapter. Also compare the number of effective samples. Can you explain the differences?

Hard.

12H1. In 1980, a typical Bengali woman could have 5 or more children in her lifetime. By the year 200, 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 12.42

```
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
```

District 54 is absent. So `district` isn't yet a good index variable, because it's not contiguous. This is easy to fix. Just make a new variable that is contiguous. This is enough to do it:

R code 12.43

```
d$district_id <- as.integer(as.factor(d$district))
sort(unique(d$district_id))
```

```
 [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 54 55 56 57 58 59 60
```

Now there are 60 values, contiguous integers 1 to 60.

Now, focus on predicting `use.contraception`, clustered by `district_id`. Do not include `urban` just yet. Fit both (1) a traditional fixed-effects model that uses dummy variables for district and (2) a multilevel model with varying intercepts for district. Plot the predicted proportions of women in each district using contraception, for both the fixed-effects model and the varying-effects model. That is, make a plot in which district ID is on the horizontal axis and expected proportion using contraception is on the vertical. Make one plot for each model, or layer them on the same plot, as you prefer. How do the models disagree? Can you explain the pattern of disagreement? In particular, can you explain the most extreme cases of disagreement, both why they happen where they do and why the models reach different inferences?

12H2. Return to the Trolley data, `data(Trolley)`, from Chapter 11. Define and fit a varying intercepts model for these data. Cluster intercepts on individual participants, as indicated by the unique values in the `id` variable. Include `action`, `intention`, and `contact` as ordinary terms. Compare the varying intercepts model and a model that ignores individuals, using both WAIC and posterior predictions. What is the impact of individual variation in these data?

12H3. The Trolley data are also clustered by `story`, which indicates a unique narrative for each vignette. Define and fit a cross-classified varying intercepts model with both `id` and `story`. Use the same ordinary terms as in the previous problem. Compare this model to the previous models. What do you infer about the impact of different stories on responses?