**The full data model**

Imagine we have a a very simple population of items with values normally distributed members with standard deviation known to be 2,



To complete the Bayesian model, we’ll assume a standard normal prior on ,



Now we’re not going to observe all y_n, but only a sample of the elements. If the model is correct, our inferences will be calibrated in expection given a random sample of items y_nfrom the population.

**Missing data**

Now let’s assume the sample of y_nwe observe is not drawn at random from the population. Imagine instead that we have a subset of Nitems from the population, and for each item n, there is a probability that the item will be included in the sample. We’ll take the log odds of inclusion to be equal to the item’s value,

. 

Now when we collect our sample, we’ll do something like poll N=2000 people from the population, but each person Pi nonly has with observations missing.

This situation arises in surveys, where non-response can bias results without careful adjustment (e.g., see Andrew’s post on pre-election polling,).

So how do we do the careful adjustment?

**Approach 1: Weighted likelihood**

A traditional approach is to inverse weight the log likelihood terms by the inclusion probability,



Thus if an item has a 20% chance of being included, its weight is 5.

In Stan, we can code the weighted likelihood as follows (assuming pi is given as data).

for (n in 1:N\_obs)

target += inv(pi[n]) \* normal\_lpdf(y[n] | mu, 2);

If we optimize with the weighted likelihood, the estimates are unbiased (i.e., the expectation of the estimate is the true value ). This is borne out in simulation.

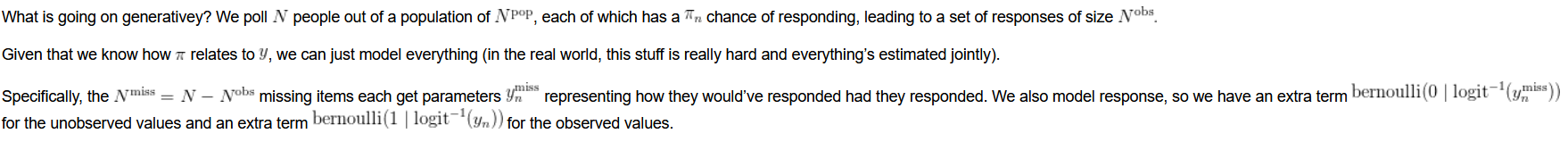
Although the parameter estimates are unbiased, the same cannot be said of the uncertainties. The posterior intervals are too narrow.

One reason the intervals are too narrow is that we are weighting the data as if we had observed Nitems when we’ve only observed items. That is, their weights are what we’d expect to get if we’d observed Nitems.

So my next thought was to standardize. Let’s take the inverse weights and normalize so the sum of inverse weights is equal to That also fails. The posterior intervals are still too narrow under simulation.

Sure, we could keep fiddling weights in an ad hoc way for this problem until they were better calibrated empirically, but this is clearly the wrong approach. We’re Bayesians and should be thinking generatively. Maybe that’s why Lauren and Andrew kept telling me I should be thinking generatively (even though they work on a survey weighting project!).

**Approach 2: Missing data**



This works. Here’s the Stan program.

data {

int N\_miss;

int N\_obs;

vector[N\_obs] y\_obs;

}

parameters {

real mu;

vector[N\_miss] y\_miss;

}

model {

// prior

mu ~ normal(0, 1);

// observed data likelihood

y\_obs ~ normal(mu, 2);

1 ~ bernoulli\_logit(y\_obs);

// missing data likelihood and missingness

y\_miss ~ normal(mu, 2);

0 ~ bernoulli\_logit(y\_miss);

}

The Bernoulli sampling statements are vectorized and repeated for each element of y\_obs and y\_miss. The suffix \_logit indicates the argument is on the log odds scale, and could have been written:

for (n in 1:N\_miss)

0 ~ bernoulli(y\_miss[n] | inv\_logit(y\_miss[n]))

And here’s the simulation code, including a cheap run at SBC:

library(rstan)

rstan\_options(auto\_write = TRUE)

options(mc.cores = parallel::detectCores(), logical = FALSE)

printf <- function(msg, ...) { cat(sprintf(msg, ...)); cat("\n") }

inv\_logit <- function(u) 1 / (1 + exp(-u))

printf("Compiling model.")

model <- stan\_model('missing.stan')

for (m in 1:20) {

# SIMULATE DATA

mu <- rnorm(1, 0, 1);

N\_tot <- 1000

y <- rnorm(N\_tot, mu, 2)

z <- rbinom(N\_tot, 1, inv\_logit(y))

y\_obs <- y[z == 1]

N\_obs <- length(y\_obs)

N\_miss <- N\_tot - N\_obs

# COMPILE AND FIT STAN MODEL

fit <- sampling(model,

data = list(N\_miss = N\_miss, N\_obs = N\_obs, y\_obs = y\_obs),

chains = 1, iter = 5000, refresh = 0)

mu\_ss <- extract(fit)$mu

mu\_hat <- mean(mu\_ss)

q25 <- quantile(mu\_ss, 0.25)

q75 <- quantile(mu\_ss, 0.75)

printf("mu = %5.2f in 50pct(%5.2f, %5.2f) = %3s; mu\_hat = %5.2f",

mu, q25, q75, ifelse(q25 <= mu && mu <= q75, "yes", "no"), mean(mu\_ss))

}

Here's some output with random seeds, with mu, mu\_hat and 50% intervals and indicator of whether mu is in the 50% posterior interval.

mu = 0.60 in 50pct( 0.50, 0.60) = no; mu\_hat = 0.55

mu = -0.73 in 50pct(-0.67, -0.56) = no; mu\_hat = -0.62

mu = 1.13 in 50pct( 1.00, 1.10) = no; mu\_hat = 1.05

mu = 1.71 in 50pct( 1.67, 1.76) = yes; mu\_hat = 1.71

mu = 0.03 in 50pct(-0.02, 0.08) = yes; mu\_hat = 0.03

mu = 0.80 in 50pct( 0.76, 0.86) = yes; mu\_hat = 0.81

The only problem I'm having is that this crashes RStan 2.19.2 on my Mac fairly regularly.

**Exercise**

How would the generative model differ if we polled members of the population at random until we got 1000 respondents? Conceptually it's more difficult in that we don't know how many non-resondents were approached on the way to 1000 respondents. This would be tricky in Stan as we don't have discrete parameter sampling---it'd have to be marginalized out.

Lauren started this conversation saying it would be hard. It took me several emails, part of a Stan meeting, buttonholing Andrew to give me an interesting example to test, lots of coaching from Lauren, then a day of working out the above simulations to convince myself the weighting wouldn't work and code up a simple version that would work.