Bayesian Cities Model 1D

Dina Sinclair

2018-12-31

Overview

This file walks through citiesMainCode1D.R.

Imagine the following: as a policymaker, you want to implement a new program in a subset of cities. To improve your knowledge of which cities would benefit the most from the program, you have the budget to run a pilot in some of the potential city choices, but not all of them. Given that you have pre-existing beliefs (priors) on how well you think the program will fare in each city, where should you run the pilot to glean the most information?

Here, we tackle this question using a bayseian hierarchical model coded in a combination of R and Stan.

Tools

This code is written in R 3.3.2 using R Markdown. The bayesian modeling is coded up in a separate language called Stan. Stan's specific tools for bayesian inference and modeling makes bayesian methods far more approachable than before. Here, we're using the rstan library to allow us to interface easily between the main R code and the bayesian Stan files.

```r
library("rstan")
```

Constants

Before we input data into the model, we need to set a few key constants. These are:

- I, the total number of cities we have priors on and are considering for the final program
- *num_pilots*, the number of cities we can run a pilot program in
- *num_final_cities*, the number of cities we can implement the final program in
- *num_draws*, the number of times we'll simulate the results of each potential pilot choice
- Q, the 'quality' of the new pilot data (the factor by which the pilot data variance will differ from the original variance). A Q of 5 implies the pilot data has a variance that is 5 times smaller than the original study variance.

```r
# Set general constants
I <- 5 # Number of cities
num_pilots <- 2 # Number of pilots
num_final_cities <- 2 # Number of cities we can implement the final program in
num_draws <- 10 # Number of times you simulate pilot study results
Q <- 5 # Pilot quality (variance multiplication factor)
set.seed(17) # Ensure randomization is reproducible through a specified seed
```

Inputs

For every city i, we need to know

- A prior Y_i that estimates the mean effect of the program on city i
- A standard error σ_i for each of these point estimates Y_i

Generating Example Inputs

In this model, we will be assuming that

$$Y_i \sim N(\theta_i, \sigma_i^2)$$

and

$$\theta_i \sim N(\mu, \tau^2)$$

To tell whether or not our model is accurately backing out μ and τ^2, we can first pick a μ and τ^2, generate input data based off of those constants, and then compare the resulting predicted μ and τ^2 program results to the constants we originally selected.

```
# Set model hyperparameters for data generation
mu <- 10
tauSq <- 2
```

To generate θ_i, we then use the fact that $\theta_i \sim N(\mu, \tau^2)$ for all i. Since the σ_i^2 are independent of our choices of μ and τ^2, we can use any σ_i^2 values we want for our generated input data. For simplicity, here we use values that are taken from U(0,1).

```
# Generate theta and sigmaSq
theta <- rnorm(I, mean = mu, sd = sqrt(tauSq)) # with theta ~ N(mu,tauSq)
sigmaSq <- 1*runif(I) # with sigmaSq ~ U(0,1)
Y <- list(mean=theta, var=sigmaSq)
```

We can then reshape the data to a more convenient structure, where Y is a list containing all of the theta and sigmaSq values. The stan file also needs all of the data in a specific format, which we're creating here in the list basic_dat_generated.

```
# Save our generated input data together in a list
basic_dat_generated <- list(I=I,Y=Y$mean,sigmaSq=Y$var)

# Display what we've generated
basic_dat_generated
```

```
## $I
## [1] 5
##
## $Y
## [1]  8.564561  9.887377  9.670507  8.844209 11.091901
##
## $sigmaSq
## [1] 0.434231178 0.002274518 0.834692139 0.830082966 0.956967818
```

Step 1: Calculate θ_i, μ, τ^2 using the original priors Y

We're assuming a **random effects model**, that is that

$$\theta_i \sim N(\mu, \tau^2)$$

and

$$Y_i \sim N(\theta_i, \sigma_i^2)$$

To make use of this assumption, we need to estimate scalars μ and τ^2 along with the vector $\theta = (\theta_1, \ldots, \theta_I)$. We can do this using stan! The following code calls up the stan model *randomEffectsModel1D.stan*, a separate text file. This code will be run in stan, with the parameters

- *iter*, the number of iterations of each chain (default is 2000)
- *chains*, the number of markov chains (default is 4)

```
fit <- stan(file = '../randomEffectsModel1D.stan',
            data = basic_dat_generated,
            iter = 1000, chains = 2)
```

```
## In file included from C:/Users/Dina/Documents/R/win-library/3.3/BH/include/boost/config.hpp:39:0,
##                  from C:/Users/Dina/Documents/R/win-library/3.3/BH/include/boost/math/tools/config.hp
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/stan/math/rev/co
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/stan/math/rev/co
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/stan/math/rev/co
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/stan/math/rev/ma
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/stan/math.hpp:4,
##                  from C:/Users/Dina/Documents/R/win-library/3.3/StanHeaders/include/src/stan/model/mo
##                  from file60c5c9f4ef3.cpp:8:
## C:/Users/Dina/Documents/R/win-library/3.3/BH/include/boost/config/compiler/gcc.hpp:186:0: warning: "
##  #  define BOOST_NO_CXX11_RVALUE_REFERENCES
##  ^
## <command-line>:0:0: note: this is the location of the previous definition
##
## SAMPLING FOR MODEL 'randomEffectsModel1D' NOW (CHAIN 1).
##
## Gradient evaluation took 0 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Adjust your expectations accordingly!
##
##
## Iteration:   1 / 1000 [  0%]  (Warmup)
## Iteration: 100 / 1000 [ 10%]  (Warmup)
## Iteration: 200 / 1000 [ 20%]  (Warmup)
## Iteration: 300 / 1000 [ 30%]  (Warmup)
## Iteration: 400 / 1000 [ 40%]  (Warmup)
## Iteration: 500 / 1000 [ 50%]  (Warmup)
## Iteration: 501 / 1000 [ 50%]  (Sampling)
## Iteration: 600 / 1000 [ 60%]  (Sampling)
## Iteration: 700 / 1000 [ 70%]  (Sampling)
## Iteration: 800 / 1000 [ 80%]  (Sampling)
## Iteration: 900 / 1000 [ 90%]  (Sampling)
## Iteration: 1000 / 1000 [100%]  (Sampling)
##
##  Elapsed Time: 0.172 seconds (Warm-up)
##                0.02 seconds (Sampling)
##                0.192 seconds (Total)
##
##
## SAMPLING FOR MODEL 'randomEffectsModel1D' NOW (CHAIN 2).
##
## Gradient evaluation took 0 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0 seconds.
## Adjust your expectations accordingly!
##
##
## Iteration:   1 / 1000 [  0%]  (Warmup)
## Iteration: 100 / 1000 [ 10%]  (Warmup)
## Iteration: 200 / 1000 [ 20%]  (Warmup)
## Iteration: 300 / 1000 [ 30%]  (Warmup)
## Iteration: 400 / 1000 [ 40%]  (Warmup)
## Iteration: 500 / 1000 [ 50%]  (Warmup)
## Iteration: 501 / 1000 [ 50%]  (Sampling)
## Iteration: 600 / 1000 [ 60%]  (Sampling)
## Iteration: 700 / 1000 [ 70%]  (Sampling)
## Iteration: 800 / 1000 [ 80%]  (Sampling)
## Iteration: 900 / 1000 [ 90%]  (Sampling)
## Iteration: 1000 / 1000 [100%]  (Sampling)
##
##  Elapsed Time: 0.191 seconds (Warm-up)
##                0.021 seconds (Sampling)
##                0.212 seconds (Total)
```

```
## Warning: There were 21 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

Sure enough, we can see that stan did two main sets of iterations (chains) adn that each of those sets was 1000 iterations long. For more details read the Stan Modeling Language User's Guide and Reference Manual section 34.

```
fit
```

```
## Inference for Stan model: randomEffectsModel1D.
## 2 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=1000.
##
##           mean se_mean   sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## mu        9.58    0.03 0.52   8.46  9.30  9.62  9.87 10.59   438 1.00
## tau       0.89    0.03 0.30   0.39  0.69  0.84  1.05  1.65   130 1.01
## theta[1]  8.90    0.04 0.46   7.99  8.58  8.89  9.22  9.82   125 1.02
## theta[2]  9.89    0.00 0.00   9.88  9.89  9.89  9.89  9.89  1000 1.00
## theta[3]  9.59    0.03 0.59   8.40  9.22  9.62  9.95 10.69   309 1.01
## theta[4]  9.25    0.03 0.61   7.93  8.87  9.33  9.69 10.31   319 1.01
## theta[5] 10.15    0.04 0.72   8.94  9.67 10.07 10.57 11.85   403 1.00
## lp__     -5.20    0.25 2.59 -10.90 -6.71 -4.99 -3.57 -0.34   111 1.01
##
## Samples were drawn using NUTS(diag_e) at Mon Dec 31 17:40:15 2018.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Looking at the data, we can see that our random effects model has generated a value for μ, τ and θ. This prediction uses the NUTS model, which is similar to the Hamiltonian Monte Carlo method (HMC) but doesn't require the user to specify a step size or number of steps.

We can now update our θ values to reflect the REM results.

```
# Readjust knowledge of Y based on REM
params <- extract(fit)
for (i in 1:length(Y$mean)){
  Y$mean[i] <- mean(params$theta[,i])
}
Y
```

```
## $mean
## [1]  8.898179  9.887380  9.585488  9.249269 10.147440
##
## $var
## [1] 0.434231178 0.002274518 0.834692139 0.830082966 0.956967818
```

Step 2: Pick a subset K studies to pilot, calculate Y_k^P

First, pick a subset of cities K to update through new pilot information. For these cities, we'll assume a **fixed effects model** and draw a new sample, imagining that for city $k \in K$ this is a new study Y_k^P done in the same place as study Y_k, so we can improve on our knowledge of Y_k. For now we will arbitrarily decide that $\sigma_k^P = \frac{1}{Q}\sigma_k$. So for $k \in K$,

$$Y_k^P \sim N(\theta_k, \frac{1}{Q}\sigma_k)$$

To do this, we'll use the following function:

```r
# Calculate new data for all K new pilots
get_pilot_results <- function(K,Y){
  # Before update, Y_P is the same as Y
  Y_P <- Y

  # Update for each new pilot k
  for (k in K){

    # Gather New Pilot Data
    new_sigmaSq <- Y$var[k] * (1/Q)
    new_mean <- rnorm( 1 , mean = Y$mean[k] , sd = sqrt(new_sigmaSq ))

    # Combine the old and new data
    post_pilot <- update_Y(Y$mean[k],new_mean,Y$var[k],new_sigmaSq)
    Y_P$mean[k] <- post_pilot$mean
    Y_P$var[k] <- post_pilot$var
    return(Y_P)
  }
}
```

Step 3: Update all Y values to get Y'

Once those k new values get calculated, update by combining to get

$$update(Y_k, Y_k^P) = Y_k'$$

This draws on the idea that given normal distributions $Y_1 \sim N(\mu_1, \sigma_1^2)$ and $Y_2 \sim N(\mu_2, \sigma_2^2)$, the two distributions can be combined to get Y' with mean μ' and variance $\sigma^{2'}$ using the equations

$$\mu' = \frac{\mu_2\sigma_1^2 + \sigma_2^2\mu_1}{\sigma_2^2 + \sigma_1^2}$$

and

$$\sigma^{2'} = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2} = \frac{1}{\frac{1}{\sigma_2^2} + \frac{1}{\sigma_1^2}}$$

Concretely, this code looks like

```r
update_Y <- function(mu1, mu2, sigSq1, sigSq2){
  update_mean  <- (mu1*sigSq2 + mu2*sigSq1)/(sigSq1 + sigSq2)
  update_var <- (sigSq1*sigSq2)/(sigSq1 + sigSq2)
  return(list(mean = update_mean , var = update_var))
}
```

Step 4: Use Y' to get final θ'

Once we have all the updated Y_i' we can use them to get θ_i' using the random effects model relationship

$$Y' \sim N(\theta_i', \sigma_i'),$$

with

$$\theta' \sim N(\mu', \tau^{2'}).$$

Using the same randomEffectsModel1D.stan file as before, we can extract that fit as fit_updated, and use it to get the needed θ'.

Once we update θ to θ', we can look at the ranking of this end result - with our new knowledge, which cities do the best? Using the order function, we can rank the cities from best (highest value for θ') to worst (lowest value for θ'). The new ranking is returned in the function below.

```r
new_ranking <- function(fit_updated,Y_P) {
  params_updated <- extract(fit_updated)
  Y_updated <- Y_P
  for (i in 1:length(Y_updated$mean)){
    Y_updated$mean[i] <- mean(params_updated$theta[,i])
  }
  new_rank <- order(Y_updated$mean, decreasing=TRUE)
  return(new_rank)
}
```

From this ranking, we next need to ask - did the information from the pilots add value? Here, we choose to define the pilot information as valuable if the act of running the pilot studies changes which subset of cities we select for the final program implementation.

In other words, we are looking at if the pilots *change our mind*. Let the number of cities that we can run the final program in be F. Then we 'change our mind' if the cities with the highest F values of θ' (the cities we'd choose after the pilots) are different than the cities that had the highest F values of θ (the cities we'd have chosen before the pilots).

The function *change_mind* below returns a boolean denoting if we will change our mind as a result of running pilots in subset of cities K. Note that this relies on the randomly generated results of the pilot studies, so sometimes a given subset of K city pilots might change our minds while other times it might not.

```r
change_mind <- function(K,Y,original_rank){
  # This function returns TRUE if the pilots run in K change the final actions
  # and returns FALSE otherwise, given original ranking and data Y.

  # Simulate pilots
  Y_P <- get_pilot_results(K,Y)
  # Update thetas
  updated_dat_generated <- list(I, Y=Y_P$mean, sigmaSq=Y_P$var)
  fit_updated <- stan(file = '../randomEffectsModel1D.stan',
                      data = updated_dat_generated,
                      iter = 1000, chains = 2)
  # Use new thetas to get a new city ranking
  new_rank <- new_ranking(fit_updated,Y_P)
  # Take the top F cities from each ranking as final city choice
  original_choice <- original_rank[1:num_final_cities]
  new_choice <- new_rank[1:num_final_cities]
  # Return the setequality of the two rankings (set bc order doesn't matter)
  return(!setequal(original_choice,new_choice))
}
```

Since the *change_mind* function returns a probabilistically generated result, we need to run the function many times to get a sense of how likely a given choice of cities $K$ will change our minds. Therefore, for each possible subset of cities $K$, we run the *change_mind* function *num_draws* times, summing up the total number of times our minds are changed.

The code for this final step is below

```
# Calculate the city ranking of the original data
original_rank <- order(Y$mean, decreasing=TRUE)
original_rank

# Generate all combinations of cities $K$, store them in vector 'combinations'
combinations <-combn(seq(I),num_pilots)
# Initiate number of minds changed (nmc) to zero
nmc <- numeric(ncol(combinations))

# Loop through all combinations K, updating nmc num_draws times
for (i in 1:ncol(combinations)){
  for (j in 1:num_draws){
    K <- combinations[,i]
    nmc[i] <- nmc[i] + change_mind(K,Y,original_rank)
  }
}
```

So what are the results? We print them here.

```
print(Y)
```

```
## $mean
## [1]  8.898179  9.887380  9.585488  9.249269 10.147440
##
## $var
## [1] 0.434231178 0.002274518 0.834692139 0.830082966 0.956967818
```

```
for (i in 1:length(nmc)){
  print(combinations[,i])
  print(paste("Number of times minds changed: ",nmc[i],"/",num_draws))
}
```

```
## [1] 1 2
## [1] "Number of times minds changed:  0 / 10"
## [1] 1 3
## [1] "Number of times minds changed:  0 / 10"
## [1] 1 4
## [1] "Number of times minds changed:  0 / 10"
## [1] 1 5
## [1] "Number of times minds changed:  0 / 10"
## [1] 2 3
## [1] "Number of times minds changed:  0 / 10"
## [1] 2 4
## [1] "Number of times minds changed:  0 / 10"
## [1] 2 5
## [1] "Number of times minds changed:  0 / 10"
## [1] 3 4
## [1] "Number of times minds changed:  1 / 10"
## [1] 3 5
## [1] "Number of times minds changed:  4 / 10"
## [1] 4 5
## [1] "Number of times minds changed:  0 / 10"
```