Population modelling for census support

Code

Tutorial 1: How to think like a Bayesian and build a first population model

Edith Darin and Douglas Leasure

Compiled on 21/08/2021

WorldPop

Introduction

The first part of this tutorial will cover how to revamp a basic frequentist model into a Bayesian mdoel and will introduce some concepts such as Directed Acyclic graph and priors.

The second part will be devoted to present the dataset we will be working with for the next three tutorials.

We will present two simple population models:

- a model based on the Normal distribution
- a multilevel model based on a Poisson-Lognormal compound.

This will be the occasion to experiment the Stan software and its R interface `rstan`.

Goals

1. Write a simple linear regression in a Bayesian framework
2. Adapt the statistician toolbox to a real-world example
3. Fit a Normal model in `stan` for modelling population:
    1. Format data for `stan`
    2. Specify a model in the `stan` language
    3. Set up a MCMC sampler to fit the model
    4. Evaluate results and limitations
4. Fit a Poisson-Lognormal model for modelling population

Supporting readings

This serie of tutorials are not an introduction to statistics. For this specific lesson, it would be good to be familiar some statistical concepts, and for that purpose we indicate useful resources:

- Probabilistic distribution
    - Linear regression (https://en.wikipedia.org/wiki/Linear_regression)
    - Normal distribution (https://en.wikipedia.org/wiki/Normal_distribution), Poisson distribution (https://en.wikipedia.org/wiki/Poisson_distribution), Log-normal distribution (https://en.wikipedia.org/wiki/Log-normal_distribution)
- [Markov chain Monte Carlo (MCMC)], a simulation-based method for model estimation:
    - Markov chains explained visually (https://setosa.io/ev/markov-chains/) by Victor Powell
    - Metropolis-Hastings Monte Carlo (https://www.bayesrulesbook.com/chapter-7.html), a specific case of MCMC that is used in `stan`. This chapter comes from the *Bayes Rules!* online book by Alicia A. Johnson, Miles Ott and Mine Dogucu
- Prior probability (https://en.wikipedia.org/wiki/Prior_probability)

And we add to that list the documentation for the software used:

- Software: Stan (https://mc-stan.org/users/documentation/)

  Stan is a C++ library for Bayesian modeling and inference that primarily uses the No-U-Turn sampler (NUTS) Hoffman and Gelman (n.d.) to obtain posterior simulations given a user-specified model and data

- Interface: rstan (https://mc-stan.org/rstan/articles/rstan.html)

  The rstan package allows one to conveniently fit Stan models from R (R Core Team 2014) and access the output, including posterior inferences and intermediate quantities such as evaluations of the log posterior density and its gradients.

From a frequentist to a Bayesian mindset

In a standard frequentist approach, a linear regression between Y the response variable and X the predictors can be formulated as:

$$Y = \alpha + \beta X + \epsilon \tag{1}$$
$$\epsilon \backsim Normal(mean = 0, sd = \sigma)$$

Equation (1) can be rewritten as:

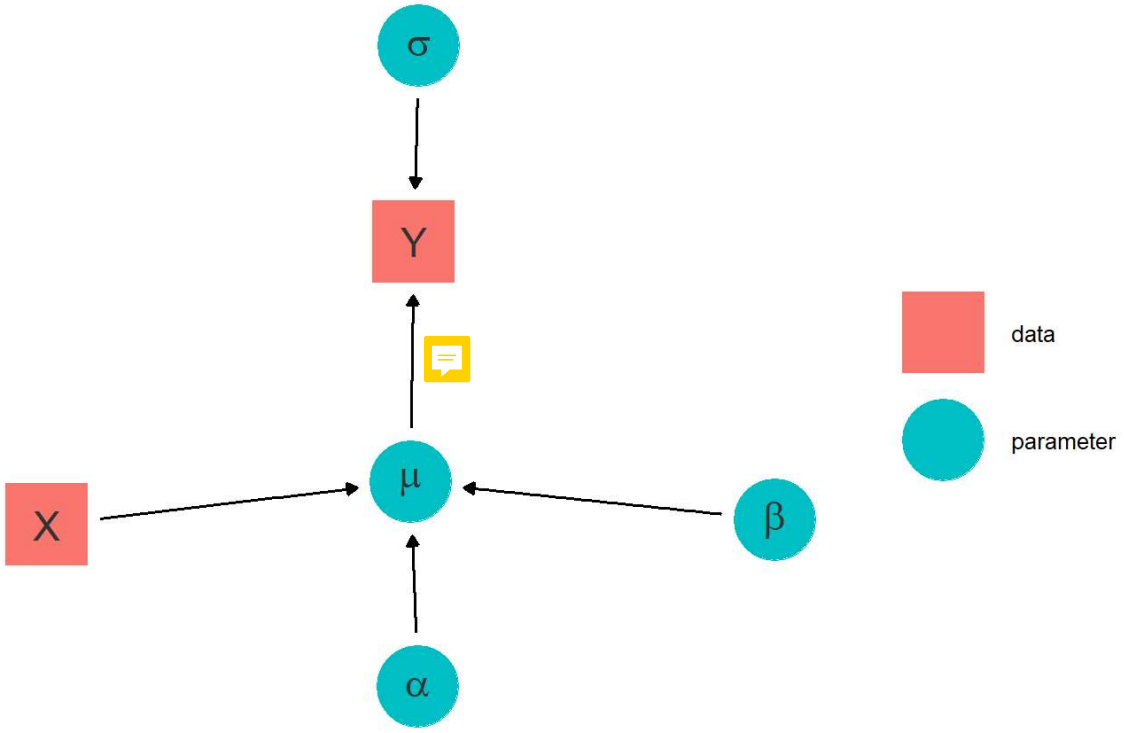
$$Y \backsim Normal(mean = \mu, sd = \sigma)$$

$$\mu = \alpha + \beta X$$

This format is more flexible when we start working with **non-normal error structures** and **custom modelling components**.

This linear regression can be represented using a directed acyclic graph (DAG) that helps to picture the relationships between model parameters and input data:

Graph of a linear regression



In a DAG, Squares represent data and circles represent parameters. Directions of arrows indicate dependence.

In a Bayesian model, all root node parameters (those with no arrows pointing towards them) need **priors** to be specified:

$$\alpha \backsim Normal(mean = 0, sd = 1000)$$

$$\beta \backsim Normal(mean = 0, sd = 1000)$$

$$\sigma \backsim Uniform(min = 0, max = 1000)$$

These are examples of **weakly informative priors** (i.e. because the means are zero and the variances are large relative to the data). Weakly informative priors should not have any noticeable effect on the final parameter estimates.

How to choose the priors TBC

To identify a distribution to use for priors, first ask yourself, "What values are possible for this parameter?"

Regression coefficients are generally continuous numbers that can take values from -∞ to +∞. The normal distribution is a good choice of prior for these parameters because it has the same characteristics. Also, for analytical purposes (having a conjugate model) and speeding up the run time, normal priors for regression coefficients are often preferred.

Standard deviations are continuous numbers that must be positive. A normal distribution is not a good choice for this prior because it includes negative numbers. A gamma distribution (positive and continuous) is a common choice of prior for a precision parameter from a normal distribution (or an inverse-Gamma as a prior for a variance parameter) because this is the conjugate prior (conjugacy speeds up the mcmc sampler…

details not important right now). But, a Gamma can be an informative prior because of the peak in probability density near zero (see Gelman 2006). Following Gelman (2006), I generally use a uniform prior distribution for standard deviation parameters.

The bayesrules book has a very good chapter (https://www.bayesrulesbook.com/chapter-4.html#) on the interaction between priors and data.

The `stan` team put together interesting guidelines (https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations) to prior setting.

Let's try with data

Set up

First, download the most recent versions of the following softwares:

- R (https://www.r-project.org/ (https://www.r-project.org/))
- RStudio (https://rstudio.com/products/rstudio/download/ (https://rstudio.com/products/rstudio/download/))

Next, install and set up the **`rstan`** package by carefully following the directions (https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started).

Hide

```
# stan setup
options(mc.cores = parallel::detectCores()) #set up the maximum number of cores used by stan
rstan::rstan_options(auto_write = TRUE) # speed up running time
```

- Install a set of data wrangling packages:

Hide

```
install.packages(c( "tidyverse", # for data manipulation
                    'kableExtra', # for good visualisation of tables
                    'here' # for handling relative path to external assets (pic, data)
                    ),
                  dependencies = TRUE)
```

Finally let's download the data we will be modelling. It belongs to the supplementary material (https://www.pnas.org/content/suppl/2020/09/09/1913050117.DCSupplemental) of the seminal paper describing WorldPop bottom-up population models (Douglas R. Leasure et al. 2020b).

Hide

```
# 2 Introduce the data ---

# download  tutorial data
download.file(
  "https://www.pnas.org/highwire/filestream/949050/field_highwire_adjunct_files/1/pnas.1913050117.sd01.xls",
  'tutorials/data/nga_demo_data.xls',
  method='libcurl',
  mode='wb'
)
```

The data

The data consists in household surveys that collected information on the total population in 1141 clusters in 15 of 37 states in Nigeria during 2016 and 2017. Clusters varied slightly in size, but were all approximately 3 hectares. These clusters were randomly sampled locations whose boundaries were drawn based on high resolution satellite imagery. The surveys are further described in Douglas R. Leasure et al. (2020a) and Weber et al. (2018). Survey sites locations are shown in Figure 1.

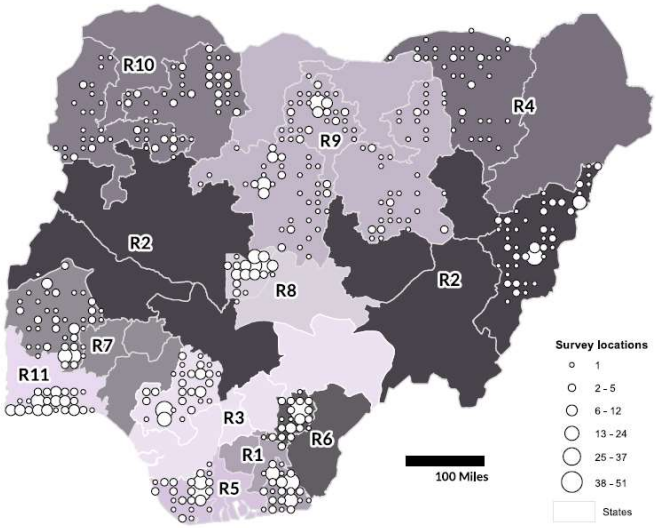


Figure 1: Map of microcensus survey as the number of survey locations within a 20 km grid cell. Region groupings are shaded and numbered R1 - R11. Source: Leasure et al. (2020).

The map in Figure 1 shows some key characteristics of the sample design:

- Only some states were sampled
- But at least 1 state per “region” was sampled
- Within states, locations were randomly sampled within settlement types

Let’s look at the table attributes:

Code

id	N	A
cluster_1	547	3.036998
cluster_2	803	2.989821
cluster_3	750	3.078278
cluster_4	281	3.019307
cluster_5	730	3.025204
cluster_6	529	3.090072
cluster_7	505	3.007513
cluster_8	402	3.019307
cluster_9	388	3.202116
cluster_10	900	3.054689

Each row is a survey site, with population counts (`N`) and the settled area (`A`) in hectares.

Response variable: the population count

We want to model the **distribution of population count** at each survey site:

Code

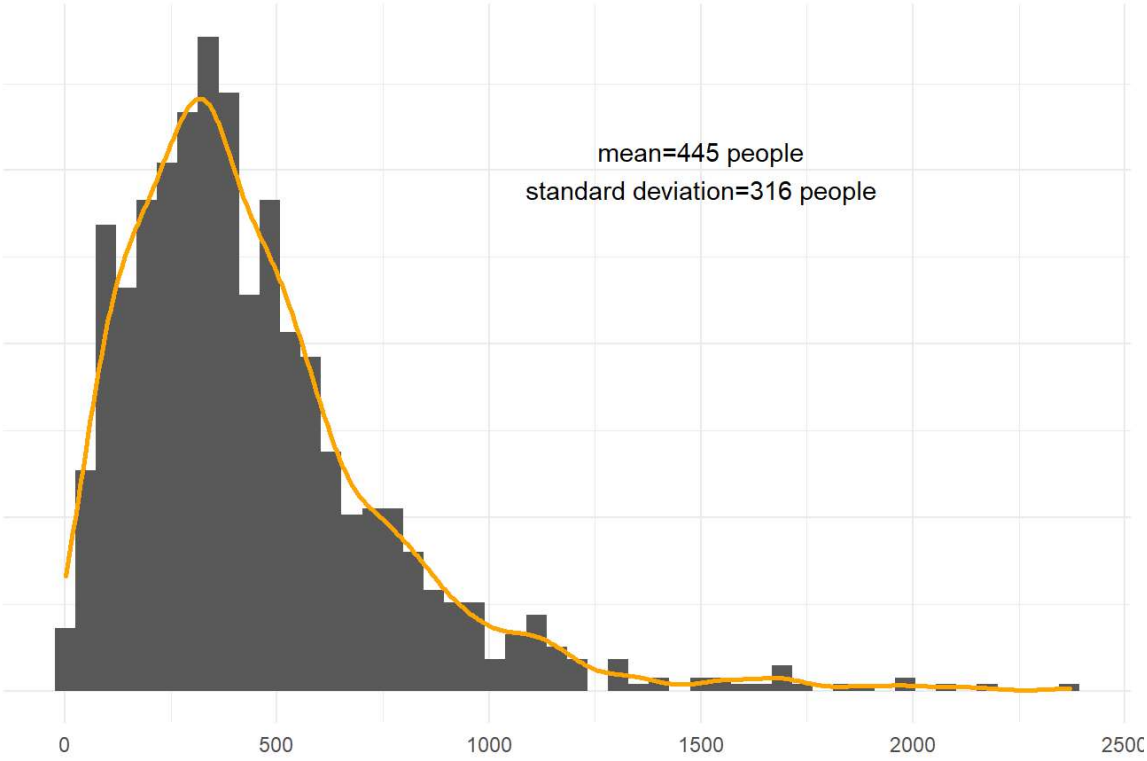


Figure 2: Observed population count distribution at survey sites

Note the wide variation in population count per survey site, with a maximum of 2370 people.

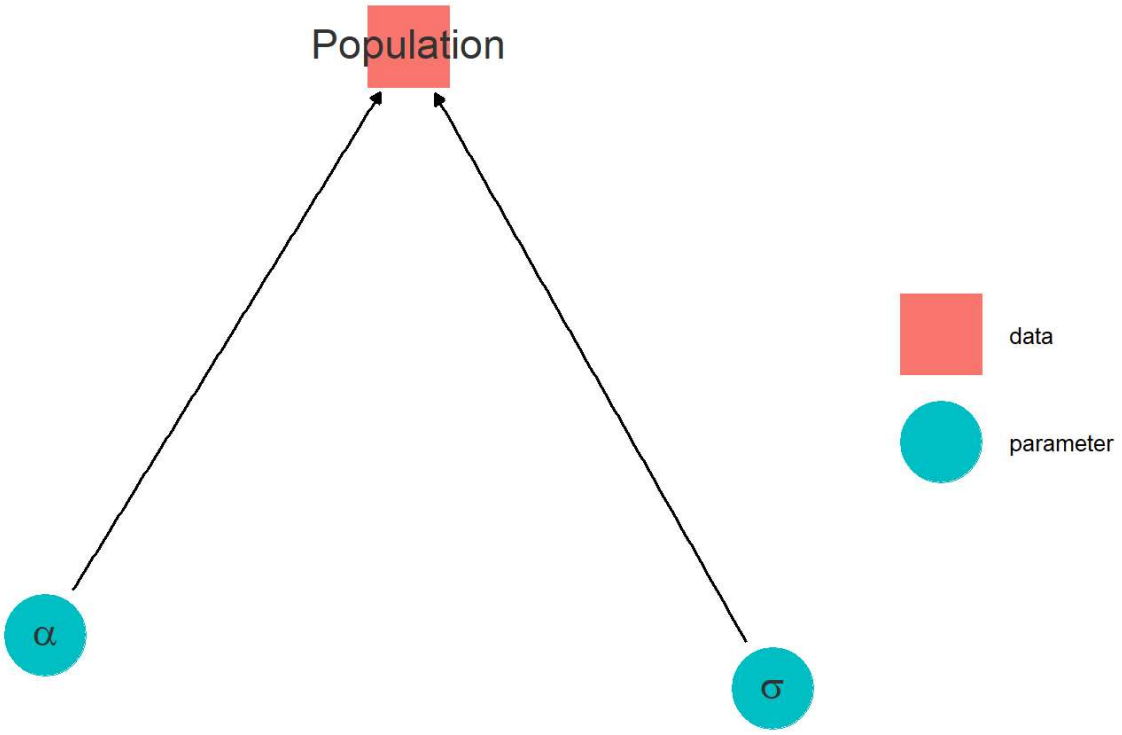
Modelling: Population count as a normal distribution

The simplest modelling assumption is that t**he population count follows a Normal distribution**. Indeed Figure 2 shows a bell-shaped curve. The parameters of a Normal distribution are the mean μ and the standard deviation σ n. Because no covariates are involved, the mean equals to the intercept of the linear regression: $\mu = \alpha$ of Equation (**??**).

$$population \sim Normal(\alpha, \sigma)$$

The corresponding DAG shows the interaction between the population count and the model parameters:

Model 1: Normal distribution of population count



We then have to define the priors of the parameters. We will choose relatively uninformed priors based on our understanding of the data.

From the observed distribution we know that the mean population count is around 450 per cluster and has a rough bell-shape. We set up the prior for α to follow a Normal centered on 450 but with wide tails to reduce the information given to the estimation process, and thus the bias introduced.

Code

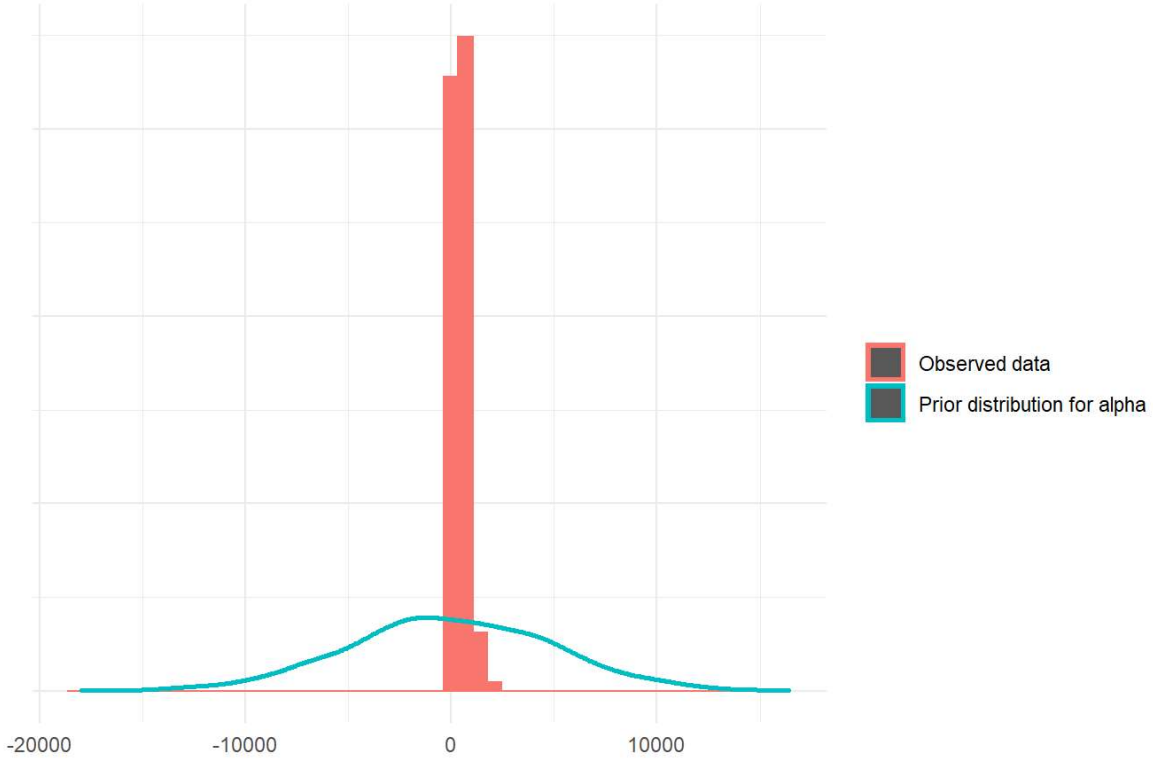


Figure 3: Prior distribution for alpha

Similarly, we know that the observed variance is around 300 people per cluster. We will set up the prior to be a uniform between 0 and 1000.

$$\begin{aligned} population &\sim Normal(\alpha, \sigma) \\ \alpha &\sim Normal(0, 5000) \\ \sigma &\sim Uniform(0, 1000) \end{aligned} \tag{2}$$

Model in `stan`

To estimate α and μ, we will write **our first model** in stan.

Hide

```
// Model 1: Population count as a normal distribution
data{
  int<lower=0> n; // number of microcensus clusters
  real<lower=0> population[n]; // count of people
}
parameters{
  // intercept
  real alpha;
  // variance
  real<lower=0> sigma;
}
model{
  // population totals
  population ~ normal( alpha, sigma );
  // intercept
  alpha ~ normal(0, 5000);
  // standard deviation
  sigma ~ uniform(0, 1000);
}
```

A `stan` model is composed of code blocks. The fundamentals ones are:

- the **data block** that describes the input data to the model
- the **parameters block** that describes the parameters to be estimated
- the **model block** that describes the stochastic elements: (1) the interaction between the parameters and the data, (2) the prior distribution

The `stan` software requires to declare all variables, both parameters and data, with their type (int, real) and size (as indicated with `[n]`)[1] . It is possible to incorporate constraints on the variable support, e.g. it is not possible to have a negative input `population` (`real<lower=0> population[n]`), and a negative σ (`real<lower=0> sigma`).

Note: `stan` requires to leave one blank line at the end of the script.

We will store the model in a `stan` file called `tutorial1_model1.stan` in the `tutorial1` folder.

Preparing data for stan

Stan software takes as input a list of the observed data that defines the variables indicated on the `data block` .

Hide

```
# prepare data for stan
stan_data <- list(
  population = data$N,
  n = nrow(data))
```

Running the model

We first set up the **parameters of the Markov Chain algorithm.**

Hide

```
# mcmc settings
chains <- 4
warmup <- 250
iter <- 500
seed <- 1789
```

The `chains` argument specified the number of Markov chains to run simultaneously. We want the markov chains to replicate a fully random process. However, the design of the chain algorithm makes every sample dependant on the previous sample. To recreate a random setting we run independently several chains to explore the parameter space and that *hopefully* converge to the same consistent solution.

The `warmup` parameter is the number of samples at the beginning of the estimation process that we discard from the results. This is similar to cooking pancakes in the sense that you need the algorithm to warm up before nearing reasonable values.

The `iter` parameter specifies the number of iterations, that is the length of the Markov chain. The longer the chain the more likely it is to stabilize around the correct estimate.

Finally we define a `seed` for the results to be exactly replicated.

Then we define the **parameters than we want to monitor,** that are stored during the estimation process:

Hide

```
# parameters to monitor
pars <- c('alpha','sigma')
```

And we are ready to **run the model**!

Hide

```
# mcmc
fit <- rstan::stan(file = file.path('tutorial1_model1.stan'),
                   data = stan_data,
                   iter = warmup + iter,
                   chains = chains,
                   warmup = warmup,
                   pars = pars,
                   seed = seed)
```

Checking the MCMC simulations

We check the Markov chains to see if they converged to a unique solution. It can be visualized with a **traceplot** for the two parameters, α and σ. A traceplot describes the evolution of the parameter estimation across the Markov chain iterations. A good traceplot sees the mixing of the different chains, evidence of convergence to a single estimate.

Hide

```
# plot trace
stan_trace(fit, inc_warmup = T)
```

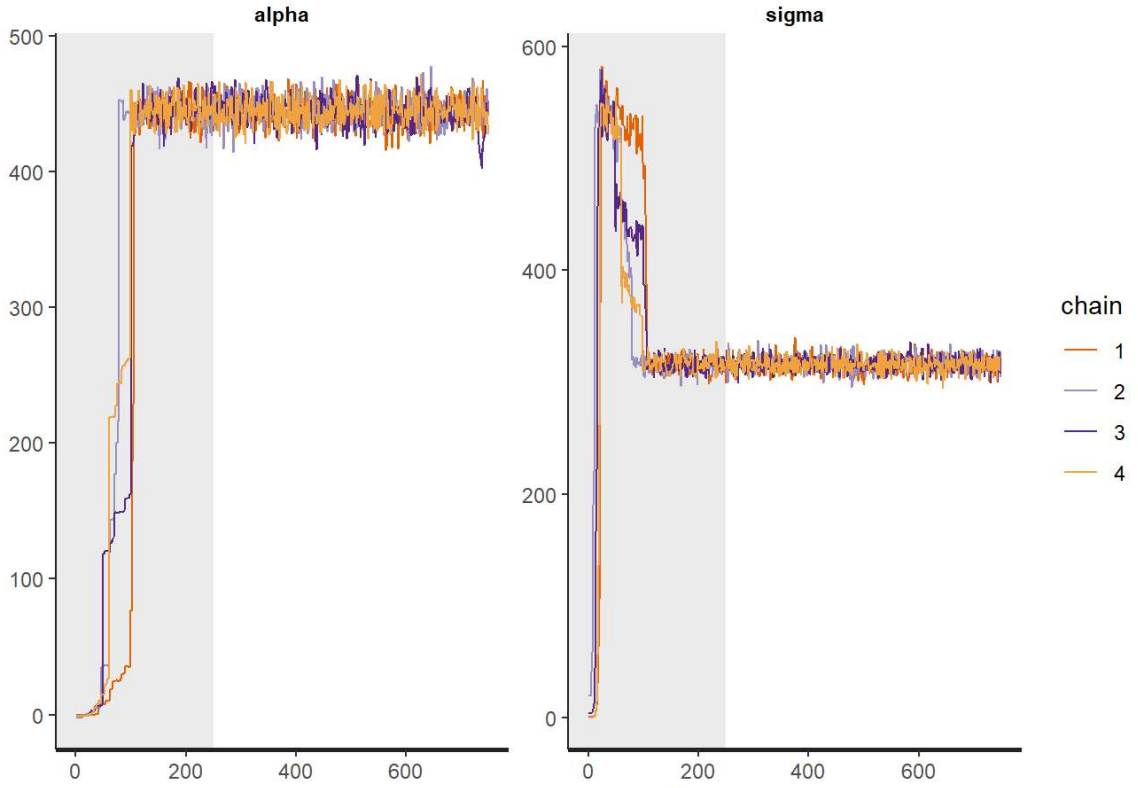


Figure 4: Model 1 Traceplot

Figure 4 shows both the warm-up period (up until 250) and the following iterations. We see that convergence happened before the end of the warm-up period and is stable over the iterations.

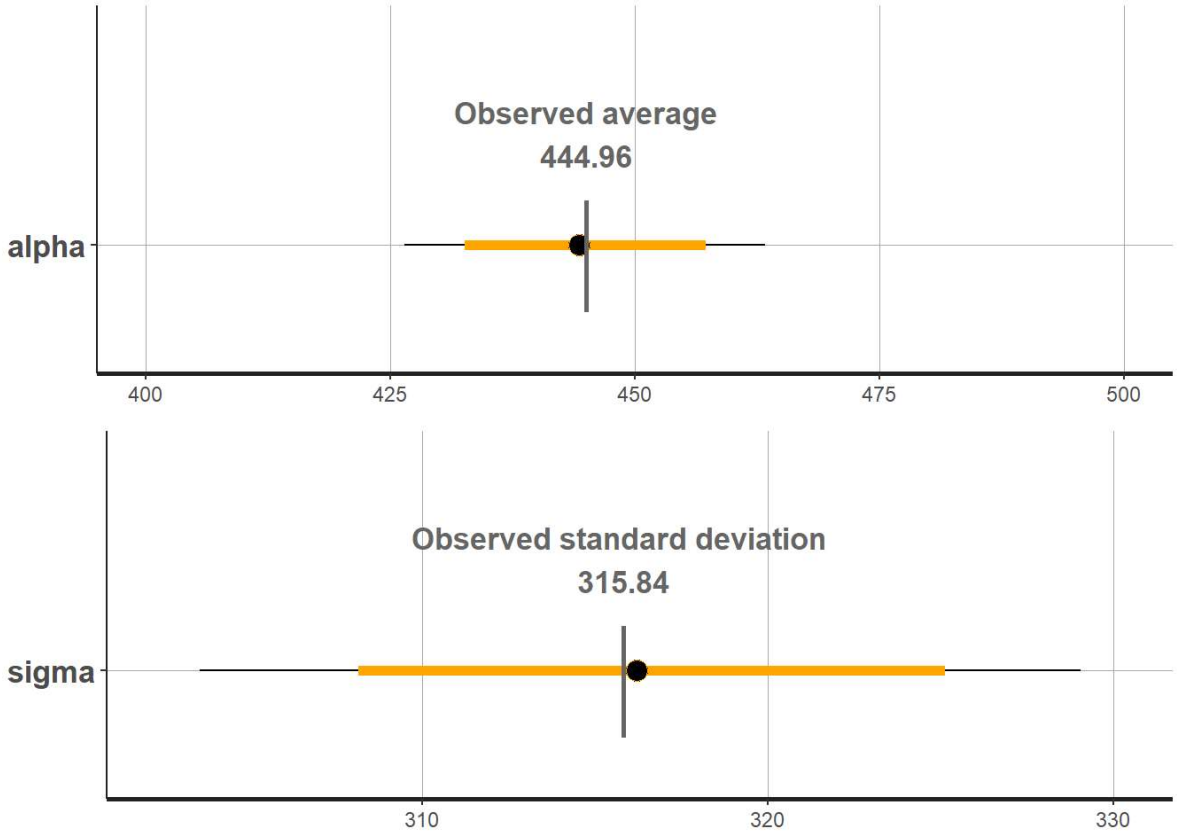
From Figure 4 (and the absence of warning from `stan`), we can conclude that the model converged and thus its goodness-of-fit can be evaluated.

Evaluating the model goodness-of-fit

Estimated parameters

We plot first **the parameters $\hat{\alpha}$ and $\hat{\sigma}$** and see how they compare with the observed average and standard deviation. The stan object named `fit` stored the estimation of the parameters at each iterations, which correspond to the *posterior distribution* of the parameters. We overlay the `stan_plot` base function with the observed mean and standard deviation.

Code



The observed variables belong to the credible interval of the estimated parameters.

Predicted population count

To see if the model is coherent with the observations, we can compute the predicted population count for every survey site. It is part of posterior predictive checking which is based on the following idea: *if a model is a good fit then we should be able to use it to generate data that looks a lot like the data we observed*.

It is possible in `stan` to do it as part of the estimation process through the **generated quantities block** .

Hide

```
// Model 1bis: Population count as a normal distribution with integrated predictions
...
generated quantities{
   real population_hat[n];

   for(idx in 1:n){
     population_hat[idx] = normal_rng( alpha, sigma );
   }
}
```

We define the parameter `population_hat` as a draw (as represented by the suffix `rng` for *random number generator*) from a Normal distribution with the estimated $\hat{\alpha}$ and $\hat{\sigma}$ at each iteration.

We re-run the model:

Hide

```
# parameter to monitor
pars <- c('alpha','sigma', 'population_hat')

# mcmc
fit_model1 <- rstan::stan(file = file.path('./tutorial1_model1bis.stan'),
                          data = stan_data,
                          iter = warmup + iter,
                          chains = chains,
                          warmup = warmup,
                          pars = pars,
                          seed = seed)
```

And extract the predicted population count.

Hide

```
# extract predictions
predicted_pop_model1 <- as_tibble(extract(fit_model1, 'population_hat')$population_hat)

colnames(predicted_pop_model1) <- data$id
```

We obtain a table with 500 predictions * 4 chains for each survey site.

Code

iteration	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7	cluster_8	cluster_9	cluster_10
iter_1	366.5476	792.5161	308.22815	-156.61490	527.8297	52.81157	552.0308	308.57074	207.95427	781.59126
iter_2	328.1199	282.8879	749.96531	677.61377	331.5469	280.02537	873.8279	697.23124	40.37625	527.20098
iter_3	838.3816	675.7826	779.17316	424.28714	758.1018	16.60610	324.6885	137.65718	534.11451	1013.68478
iter_4	789.9939	293.2120	-81.12327	935.68203	486.1098	118.60275	353.9939	98.33023	693.73047	1247.88323
iter_5	359.5756	642.9399	-41.42014	288.11911	729.9181	391.59217	756.5040	490.11845	658.68959	186.79392
iter_6	651.7410	421.0562	655.24791	104.38340	534.8909	-155.24391	419.0243	254.54908	689.48470	172.94602
iter_7	738.9178	359.5573	321.65558	742.13224	1063.4171	756.57082	337.3649	752.37561	233.61020	895.58088
iter_8	687.9566	-113.8322	-164.01277	-43.18685	636.9383	294.87613	380.6126	55.45636	111.36183	376.14917
iter_9	772.1279	468.8460	303.80464	586.07991	-128.1306	202.43753	469.8588	66.65152	124.80616	69.29444
iter_10	383.2533	178.2390	120.92495	45.33671	738.5616	661.67488	255.4246	619.70896	491.39701	748.33273

We get thus a **posterior prediction distribution of population count** for every survey site. Figure 5 shows a posterior distribution for the first survey site.

Code

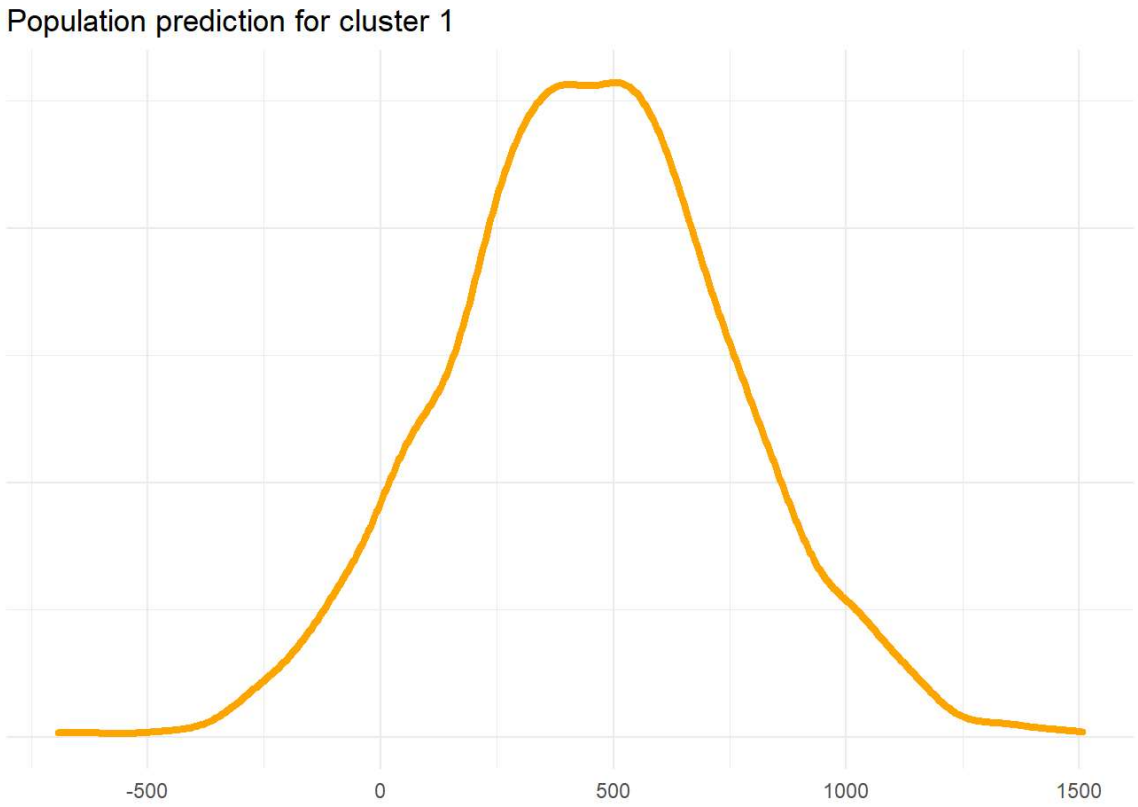


Figure 5: Example of posterior prediction distribution for one cluster

We can extract for every survey site its mean prediction and 95% credible interval.

Hide

```
# summarize predictions
comparison_df <- predicted_pop_model1 %>%
      pivot_longer(everything(),names_to = 'id', values_to = 'predicted') %>%
      group_by(id) %>%
      summarise(across(everything(), list(mean=~mean(.),
                                          upper=~quantile(., probs=0.975),
                                          lower=~quantile(., probs=0.025))))
comparison_df %>% head() %>% kbl() %>% kable_minimal()
```

id	predicted_mean	predicted_upper	predicted_lower
cluster_1	450.2411	1065.360	-148.6820
cluster_10	448.7399	1058.319	-183.9011
cluster_100	435.2787	1046.388	-147.6226
cluster_1000	437.1025	1017.653	-169.6521
cluster_1001	443.9021	1051.133	-156.1243
cluster_1002	431.1784	1031.666	-199.2759

We note that some predictions are negative - this is due to choosing a Normal distribution for representing the population count.

Let's see the global picture by plotting the observed vs the predicted population count. A perfect model would see all points on the 1:1 line.

Code

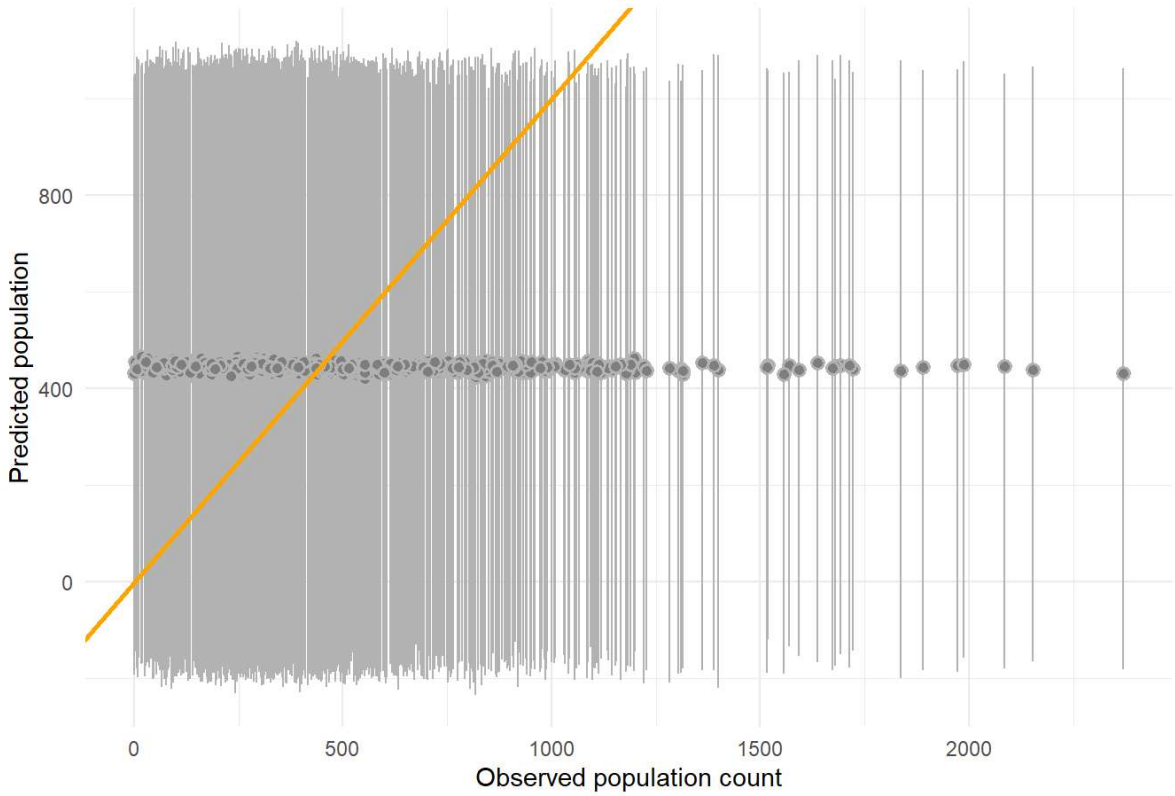


Figure 6: Comparison between observed and predicted population count (with the Normal model). Orange line indicates the 1:1 line

Figure 6 is a great visualization of the prediction process. Since the model has no covariates (introduced in tutorial 3) and no hierarchical structure (introduced in tutorial 2), the **predictions are drawn from the exact same distribution**.

We can compute goodness-of-fit metrics to complete the picture:

- The **bias,** the mean of the residuals (prediction - observation)
- The **imprecision**, standard deviation of the residual
- The **inacurracy,** mean of the absolute residuals
- The **proportion of observations falling into the predicted credible interval**
- The **r-squared**, computed as the squared correlation between predictions and observations

Hide

```
# compute goodness-of-fit metrics
comparison_df %>%
  mutate(residual = predicted_mean-N,
         in_CI = ifelse(N>predicted_lower &N<predicted_upper, T, F)) %>%
  summarise(
    `Bias`= mean(residual),
    `Imprecision` = sd(residual),
    `Inaccuracy` = mean(abs(residual)),
    `Correct credible interval (in %)` = round(sum(in_CI)/n()*100,1),
    R2 = cor(predicted_mean, N)^2
  ) %>%
    kbl(caption = "Normal model goodness-of-fit metrics") %>% kable_minimal()
```

Table 1: Normal model goodness-of-fit metrics

Bias	Imprecision	Inaccuracy	Correct credible interval (in %)	R2
-0.3364709	316.1777	231.25	95.1	0.001443

Table 1 shows that in average the predictions are off by 231 people or a 180% relative error. We see however that the model is well-specified and the uncertainty is correctly taking into account with 95% of the observations being in the 95% credible interval.

Modelling: Population count as a Poisson Lognormal model

The Normal model offers a quick solution for modelling population count (see Equation (2)). It is however unsatisfying because in reality:

- Population counts are discrete, when a Normal model assumes a continuous random variable
- Population counts are positive, when the Normal distribution's support is $-\infty$ to $+\infty$
- Observed population counts have a lot of variation due to the difference in study site size that is not taking into account in the model

We have to get back to the model itself.

Second response variable: the population density

A continuous variable that can be extracted from our data is the **population density.**

Code

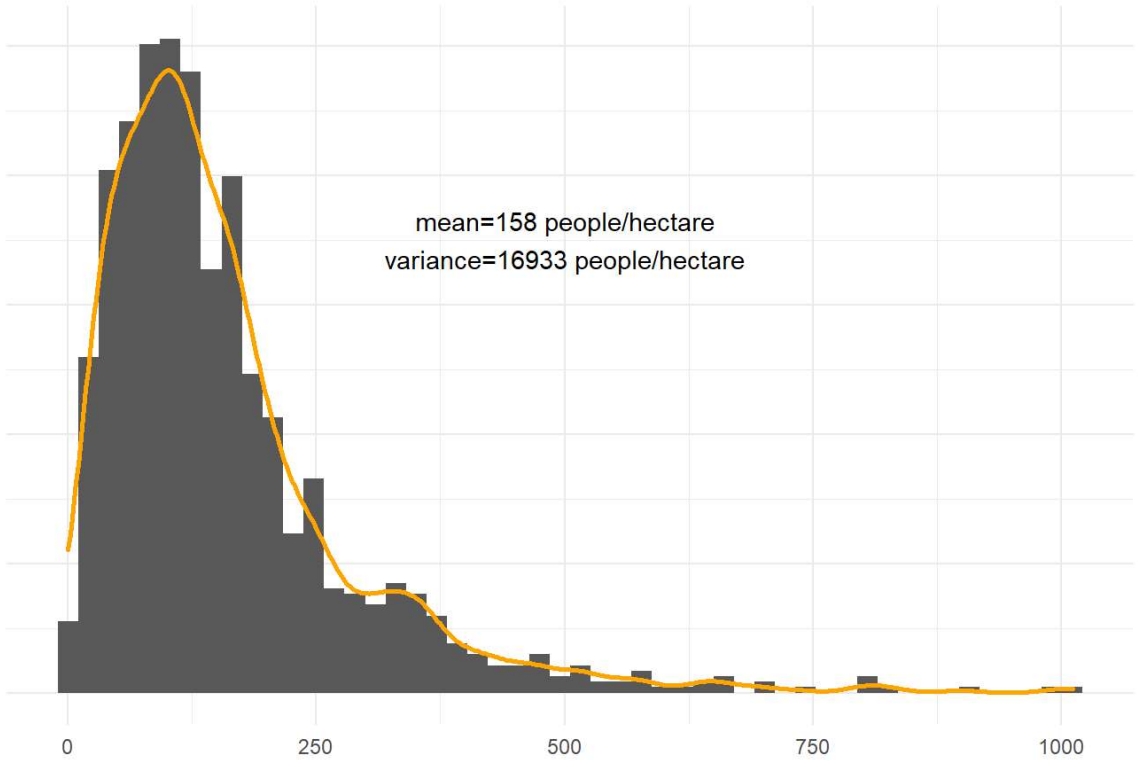


Figure 7: Observed population density distribution

Figure 8 compares the distribution of the population count and the population density. We see that the population density has a more continuous and bell-curve distribution, with a shorter right tail. The decrease in response variable variation is partly due to the heterogeneity in study site size, that are taking into account when considering the population density. Indeed study site size ranges between 0.54 and 10 hectares.

Code

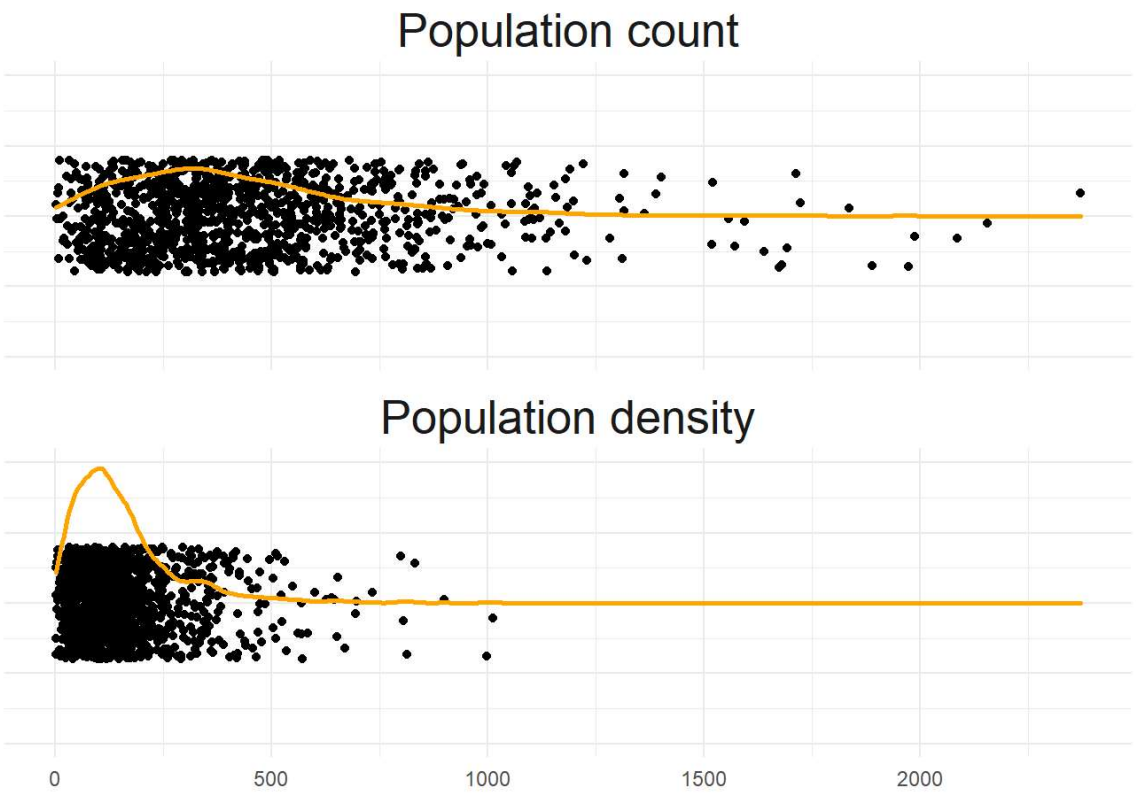


Figure 8: Response variable distribution comparaison

Poisson Lognormal model

To model population count, we use the Poisson distribution (https://en.wikipedia.org/wiki/Poisson_distribution), that describes positive discrete events.

The issue with the Poisson distribution is that it has only one single parameter, λ, that controls both the mean and the variance. It makes the model insensitive to overdispersion, that was previously captured by σ in the Normal model.

Therefore we choose the Poisson Lognormal model, where the Lognormal distribution (https://en.wikipedia.org/wiki/Log-normal_distribution) captures overdispersion and enforces a positive outcome.

$$population \backsim Poisson(\lambda)$$
$$\lambda \backsim Lognormal(\alpha, \sigma)$$

Note that this equation is equivalent to:

$$population \backsim Poisson(\lambda)$$
$$log(\lambda) \backsim Normal(\alpha, \sigma)$$

Under this form we see that the model is not linear, but log-linear. It can be rewritten as:

$$population \backsim Poisson(\lambda)$$
$$\lambda \backsim exp(Normal(\alpha, \sigma))$$

In our particular case, we decompose λ as the $population\_density * settled\_area$ which introduces the continuous variable, $population\_density$.

In the Lognormal, the parameter α represents the median of the population density on the log scale, and σ the geometric standard deviation of population density on the log scale.

We set up their priors similarly as before and retrieve from the data that the log observed mean is 5.93 and the observed log geometric standard deviation is 0.87.

$$population \backsim Poisson(pop\_density * settled\_area) \quad (3)$$
$$pop\_density \backsim Lognormal(\alpha, \sigma)$$
$$\alpha \backsim Normal(0, 100)$$
$$\sigma \backsim Uniform(0, 100)$$

We adapt the `stan` code to the model change which affects all code blocks:

Hide

```
// Model 2: Population count as a Poisson-Lognormal distribution
data{
  int<lower=0> n; // number of microcensus clusters
  int<lower=0> population[n]; // count of people
  vector<lower=0>[n] area; // settled area
}
parameters{
  // population density
  vector<lower=0>[n] pop_density;
  // intercept
  real alpha;
  // variance
  real<lower=0> sigma;
}
model{
  // population totals
  population ~ poisson(pop_density .* area);
  pop_density ~ lognormal( alpha, sigma );
  // intercept
  alpha ~ normal(0, 100);
  // variance
  sigma ~ uniform(0, 100);
}
generated quantities{
   int<lower=0> population_hat[n];
   real<lower=0> density_hat[n];

   for(idx in 1:n){
     density_hat[idx] = lognormal_rng( alpha, sigma );
     population_hat[idx] = poisson_rng(density_hat[idx] * area[idx]);
   }
}
```

We store the model under `tutorial1_model2.stan`, and prepare the corresponding data:

Hide

```
# prepare data for stan
stan_data_model2 <- list(
  population = data$N,
  n = nrow(data),
  area = data$A)
```

Then we declare the parameters to monitor (including `density_hat`) and run the model.

Hide

```
# set parameters to monitor
pars <- c('alpha','sigma', 'population_hat', 'density_hat')

# mcmc
fit_model2 <- rstan::stan(file = file.path('tutorial1_model2.stan'),
                  data = stan_data_model2,
                  iter = warmup + iter,
                  chains = chains,
                  warmup = warmup,
                  pars = pars,
                  seed = seed)
```

```
## Warning in .local(object, ...): some chains had errors; consider specifying
## chains = 1 to debug
```

```
## here are whatever error messages were returned
```

```
## [[1]]
## Stan model 'tutorial1_model2' does not contain samples.
##
## [[2]]
## Stan model 'tutorial1_model2' does not contain samples.
```

```
## Warning in validityMethod(object): The following variables have undefined
## values: population_hat[1],The following variables have undefined
## values: population_hat[2],The following variables have undefined
## values: population_hat[3],The following variables have undefined
## values: population_hat[4],The following variables have undefined
## values: population_hat[5],The following variables have undefined
## values: population_hat[6],The following variables have undefined
## values: population_hat[7],The following variables have undefined
## values: population_hat[8],The following variables have undefined
## values: population_hat[9],The following variables have undefined values:
## population_hat[10],The following variables have undefined values:
## population_hat[11],The following variables have undefined values:
## population_hat[12],The following variables have undefined values:
## population_hat[13],The following variables have undefined values:
## population_hat[14],The following variables have undefined values:
## population_hat[15],The following variables have undefined values:
## population_hat[16],The following variables have undefined values:
## population_hat[17],The following variables have undefined values:
## population_hat[18],The following variables have undefined values:
## population_hat[19],The following variables have undefined values:
## population_hat[20],The following variables have undefined values:
## population_hat[21],The following variables have undefined values:
## population_hat[22],The following variables have undefined values:
## population_hat[23],The following variables have undefined values:
## population_hat[24],The following variables have undefined values:
## population_hat[25],The following variables have undefined values:
## population_hat[26],The following variables have undefined values:
## population_hat[27],The following variables have undefined values:
## population_hat[28],The following variables have undefined values:
## population_hat[29],The following variables have undefined values:
## population_hat[30],The following variables have undefined values:
## population_hat[31],The following variables have undefined values:
## population_hat[32],The following variables have undefined values:
## population_hat[33],The following variables have undefined values:
## population_hat[34],The following variables have undefined values:
## population_hat[35],The following variables have undefined values:
## population_hat[36],The following variables have undefined values:
## population_hat[37],The following variables have undefined values:
## population_hat[38],The following variables have undefined values:
## population_hat[39],The following variables have undefined values:
## population_hat[40],The following variables have undefined values:
## population_hat[41],The following variables have undefined values:
## population_hat[42],The following variables have undefined values:
## population_hat[43],The following variables have undefined values:
## population_hat[44],The following variables have undefined values:
## population_hat[45],The following variables have undefined values:
## population_hat[46],The following variables have undefined values:
## population_hat[47],The following variables have undefined values:
## population_hat[48],The following variables have undefined values:
## population_hat[49],The following variables have undefined values:
## population_hat[50],The following variables have undefined values:
## population_hat[51],The following variables have undefined values:
## population_hat[52],The following variables have undefined values:
## population_hat[53],The following variables have undefined values:
## population_hat[54],The following variables have undefined values:
## population_hat[55],The following variables have undefined values:
## population_hat[56],The following variables have undefined values:
```

```
## population_hat[57],The following variables have undefined values:
## population_hat[58],The following variables have undefined values:
## population_hat[59],The following variables have undefined values:
## population_hat[60],The following variables have undefined values:
## population_hat[61],The following variables have undefined values:
## population_hat[62],The following variables have undefined values:
## population_hat[63],The following variables have undefined values:
## population_hat[64],The following variables have undefined values:
## population_hat[65],The following variables have undefined values:
## population_hat[66],The following variables have undefined values:
## population_hat[67],The following variables have undefined values:
## population_hat[68],The following variables have undefined values:
## population_hat[69],The following variables have undefined values:
## population_hat[70],The following variables have undefined values:
## population_hat[71],The following variables have undefined values:
## population_hat[72],The following variables have undefined values:
## population_hat[73],The following variables have undefined values:
## population_hat[74],The following variables have undefined values:
## population_hat[75],The following variables have undefined values:
## population_hat[76],The following variables have undefined values:
## population_hat[77],The following variables have undefined values:
## population_hat[78],The following variables have undefined values:
## population_hat[79],The following variables have undefined values:
## population_hat[80],The following variables have undefined values:
## population_hat[81],The following variables have undefined values:
## population_hat[82],The following variables have undefined values:
## population_hat[83],The following variables have undefined values:
## population_hat[84],The following variables have undefined values:
## population_hat[85],The following variables have undefined values:
## population_hat[86],The following variables have undefined values:
## population_hat[87],The following variables have undefined values:
## population_hat[88],The following variables have undefined values:
## population_hat[89],The following variables have undefined values:
## population_hat[90],The following variables have undefined values:
## population_hat[91],The following variables have undefined values:
## population_hat[92],The following variables have undefined values:
## population_hat[93],The following variables have undefined values:
## population_hat[94],The following variables have undefined values:
## population_hat[95],The following variables have undefined values:
## population_hat[96],The following variables have undefined values:
## population_hat[97],The following variables have undefined values:
## population_hat[98],The following variables have undefined values:
## population_hat[99],The following variables have undefined values:
## population_hat[100],The following variables have undefined values:
## population_hat[101],The following variables have undefined values:
## population_hat[102],The following variables have undefined values:
## population_hat[103],The following variables have undefined values:
## population_hat[104],The following variables have undefined values:
## population_hat[105],The following variables have undefined values:
## population_hat[106],The following variables have undefined values:
## population_hat[107],The following variables have undefined values:
## population_hat[108],The following variables have undefined values:
## population_hat[109],The following variables have undefined values:
## population_hat[110],The following variables have undefined values:
## population_hat[111],The following variables have undefined values:
## population_hat[112],The following variables have undefined values:
## population_hat[113],The following variables have undefined values:
## population_hat[114],The following variables have undefined values:
```

```
## population_hat[115],The following variables have undefined values:
## population_hat[116],The following variables have undefined values:
## population_hat[117],The following variables have undefined values:
## population_hat[118],The following variables have undefined values:
## population_hat[119],The following variables have undefined values:
## population_hat[120],The following variables have undefined values:
## population_hat[121],The following variables have undefined values:
## population_hat[122],Th
```

Warnings are thrown stating that the Poisson rate is too large. This is because `stan` enforces a upper bound for the Poisson rate. Our model is likely to get over this upper bound during the warm-up period when exploring the parameter space. We could enforce constrains on the parameter but since this issue is arising only in the *generated quantities block* that is a side model block we will implement a little trick:

Hide

```
// Model 2: Population count as a Poisson-Lognormal distribution
generated quantities{
  ...
  for(idx in 1:n){
    density_hat[idx] = lognormal_rng( alpha, sigma );

    if(density_hat[idx] * area[idx]<1e+09){
      population_hat[idx] = poisson_rng(density_hat[idx] * area[idx]);
    } else {
      population_hat[idx] = -1;
    }
  }
}
```

We re-run the model with the trick:

Hide

```
# mcmc
fit_model2 <- rstan::stan(file = file.path('tutorial1_model2bis.stan'),
                  data = stan_data_model2,
                  iter = warmup + iter,
                  chains = chains,
                  warmup = warmup,
                  pars = pars,
                  seed = seed)
```

No warnings are shown.

Question: Can you plot the traceplot and interpret it?

▶

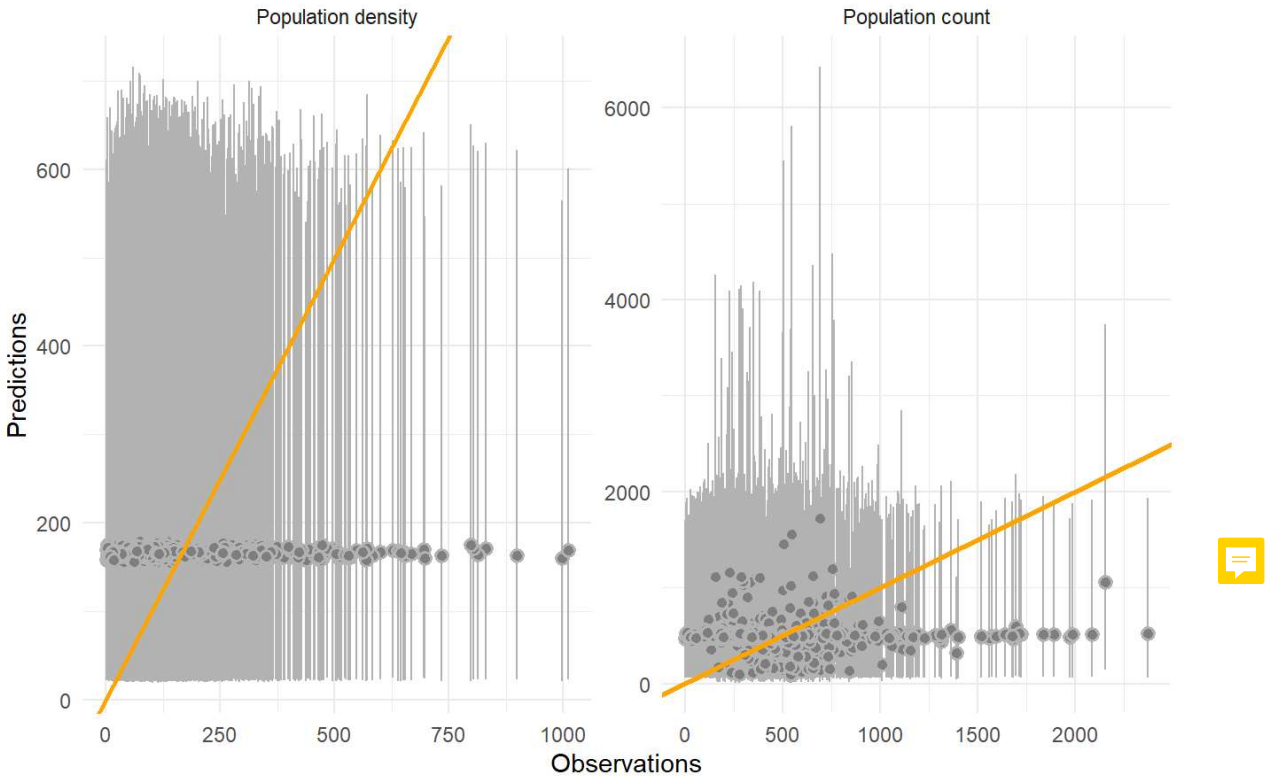
Click for the solution

We plot the predicted density and the predicted count against the observations:

Code

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

Code



We see for the `population_density` the same estimation pattern than in the Normal model, that is a similar posterior prediction distribution for every survey sites. The predicted `population_count` is, in contrast, influenced by the `settled_area` .

The goodness-of-fit of the Poisson-Lognormal distribution is however not good:

Code

Table 2: Poisson-Lognormal model goodness-of-fit metrics

Bias	Imprecision	Inaccuracy	Correct credible interval (in %)	R2
61.83258	337.7841	265.7909	94.9	0.0002384

It will require further refinements that will be introduced in the next tutorials.

And for that purpose we will store this last model, as an RDS file.

Hide

```
saveRDS(fit_model2, 'tutorial1_model2_fit.rds')
```

Bonus: What happens with a wrong prior specification?

Specifying credible priors is an art in Bayesian analysis. Let's seen what happens when we set unrealistic prior, typically priors that have a support too constrained for the data.

Let's look back at the Normal model presented in Equation (2) and choose very constrained priors, for example:

$$population \backsim Normal(\alpha, \sigma)$$
$$\alpha \backsim Normal(0, 15)$$
$$\sigma \backsim Uniform(0, 5)$$

We store this model under `tutorial1_model1wrong.stan` and run it:

Hide

```
# 5 Test prior ---
# set parameters to monitor
pars <- c('alpha','sigma')

# mcmc
fit_wrong <- rstan::stan(file = file.path('tutorial1_model1wrong.stan'),
                   data = stan_data,
                   iter = warmup + iter,
                   chains = chains,
                   warmup = warmup,
                   pars = pars,
                   seed = seed)
```

```
## Warning: There were 1758 divergent transitions after warmup. See
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
## to find out why this is a problem and how to eliminate them.
```

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

```
## Warning: The largest R-hat is 3.36, indicating chains have not mixed.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#r-hat
```

```
## Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior means and medians
may be unreliable.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#bulk-ess
```

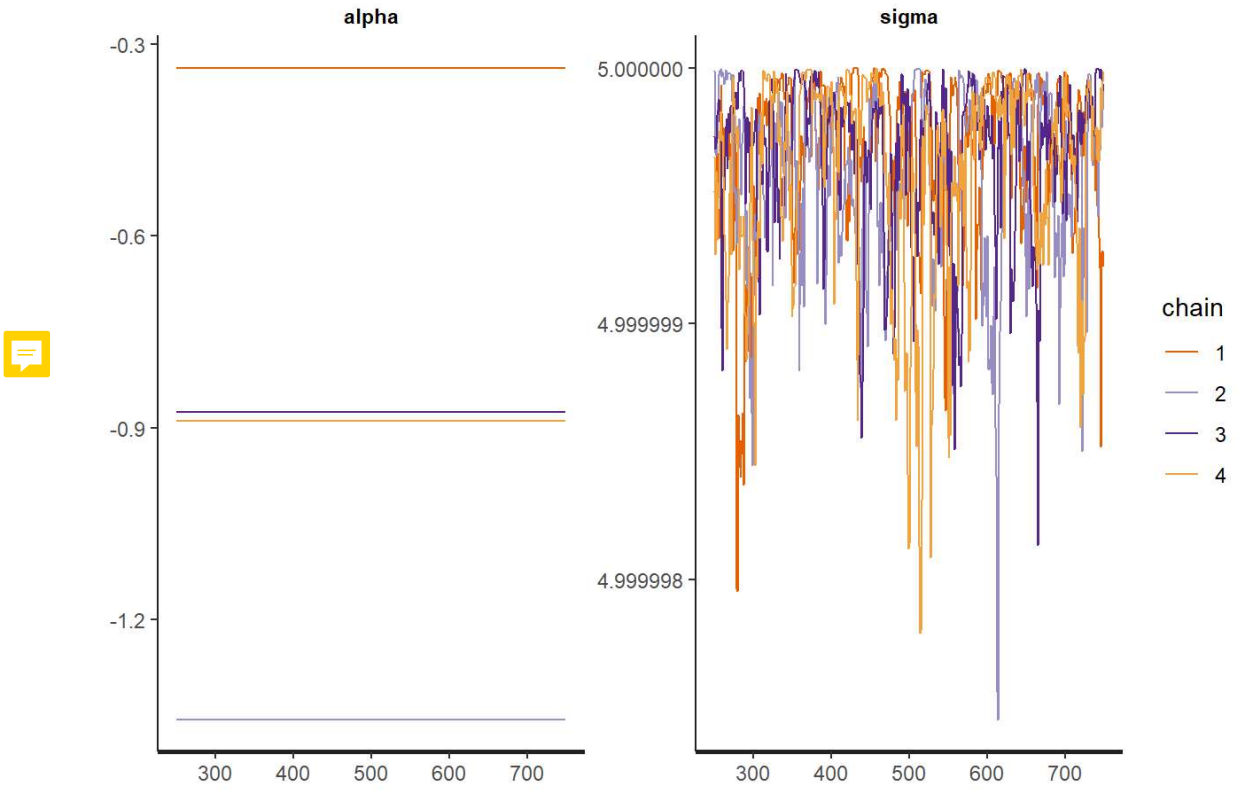
```
## Warning: Tail Effective Samples Size (ESS) is too low, indicating posterior variances and tai
l quantiles may be unreliable.
## Running the chains for more iterations may help. See
## http://mc-stan.org/misc/warnings.html#tail-ess
```

We see that estimating this model has led to a LOT of warnings. They are related with different diagnostics of the algorithm convergence. For a nice overview of parameters tuning please check: https://mc-stan.org/misc/warnings.html (https://mc-stan.org/misc/warnings.html)

The traceplot indicates clear convergence issue:

Hide

```
# plot trace
traceplot(fit_wrong)
```



First the estimation of `alpha` did not converge. Second the estimation of `sigma` is blocked at 5 which corresponds to the upper bound of the prior.

References

Hoffman, Matthew D, and Andrew Gelman. n.d. “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo,” 31.

Leasure, Douglas R., Warren C. Jochem, Eric M. Weber, Vincent Seaman, and Andrew J. Tatem. 2020a. “National Population Mapping from Sparse Survey Data: A Hierarchical Bayesian Modeling Framework to Account for Uncertainty.” *Proceedings of the National Academy of Sciences* 117 (39): 24173–79. https://doi.org/10.1073/pnas.1913050117 (https://doi.org/10.1073/pnas.1913050117).

Leasure, Douglas R, Warren C Jochem, Eric M Weber, Vincent Seaman, and Andrew J Tatem. 2020b. “National Population Mapping from Sparse Survey Data: A Hierarchical Bayesian Modeling Framework to Account for Uncertainty.” *Proceedings of the National Academy of Sciences*.

Weber, Eric M., Vincent Y. Seaman, Robert N. Stewart, Tomas J. Bird, Andrew J. Tatem, Jacob J. McKee, Budhendra L. Bhaduri, Jessica J. Moehl, and Andrew E. Reith. 2018. “Census-Independent Population Mapping in Northern Nigeria.” *Remote Sensing of Environment* 204 (January): 786–98. https://doi.org/10.1016/j.rse.2017.09.024 (https://doi.org/10.1016/j.rse.2017.09.024).

---

1. More details here (https://mc-stan.org/docs/2_27/reference-manual/overview-of-data-types.html).↩