Population modelling for census support

Tutorial 1: How to think population as a Bayesian?

Compiled on 15/09/2021

WorldPop

Introduction

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

The second part will be devoted to presenting 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 with the Stan software and its R interface [redacted].

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 [redacted] for modelling population:
    1. Format data for [redacted]
    2. Specify a model in the [redacted] 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 series 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
    - Normal distribution, Poisson distribution, Log-normal distribution
- Markov chain Monte Carlo (MCMC), a simulation-based method for model estimation:
    - Markov chains explained visually by Victor Powell
    - Metropolis-Hastings Monte Carlo, a specific case of MCMC that is used in [redacted]. This chapter comes from the *Bayes Rules!* online book by Alicia A. Johnson, Miles Ott and Mine Dogucu
- Prior probability

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

- Software: Stan

  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

  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:

$$\begin{aligned} Y &= \alpha + \beta X + \epsilon \\ \epsilon &\sim \mathrm{Normal}(\mathrm{mean} = 0, \mathrm{sd} = \sigma) \end{aligned} \tag{1}$$

Equation (1) can be rewritten as:

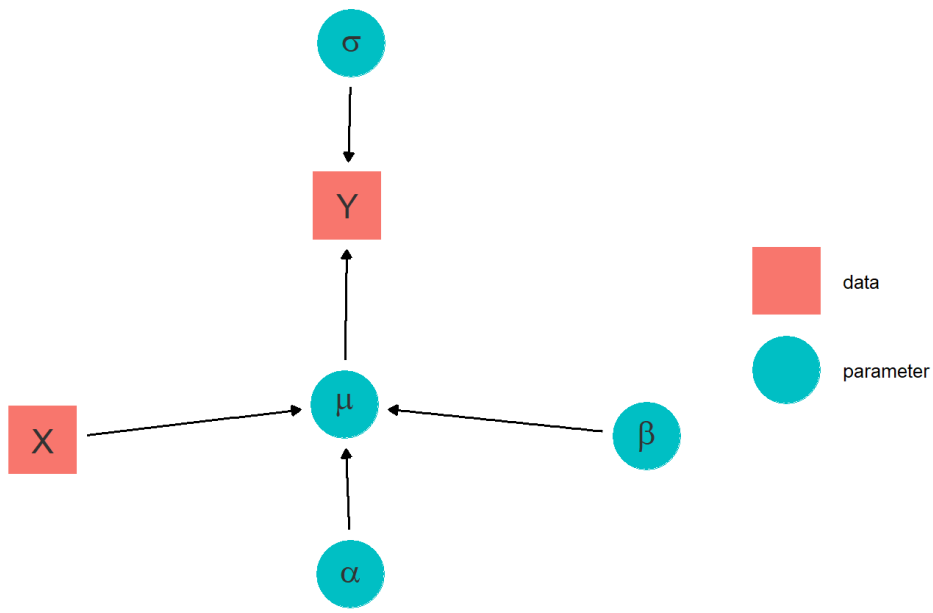
$$Y \sim \mathrm{Normal}(\mathrm{mean} = \mu, \mathrm{sd} = \sigma)$$

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

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:

$$\alpha \sim \mathrm{Normal}(\mathrm{mean} = 0, \mathrm{sd} = 1000)$$

$$\beta \sim \mathrm{Normal}(\mathrm{mean} = 0, \mathrm{sd} = 1000)$$

$$\sigma \sim \mathrm{Uniform}(\mathrm{min} = 0, \mathrm{max} = 1000)$$

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 on the interaction between priors and data.

The [redacted] team put together interesting guidelines to prior setting.

Let’s try with simulated data

Setting up

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

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

Next, install and set up the [redacted] package by carefully following the directions.

```
# 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:

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

For more information, check out their vignettes: tidyverse, kableExtra and here.

Simulating data

We will simulate **fake observations** to introduce the basic concepts of Bayesian modelling and their implementation in [redacted].

We produce our fake data as 1000 draws of a Normal with mean 5 and standard deviation 50:

```
seed <- 2004
set.seed(seed)
data <- tibble(y=rnorm(1e3, mean= 5, sd=50))
```

Note that we define a [redacted] for the results to be exactly replicated.

The observed distribution of our simulated data is the following:

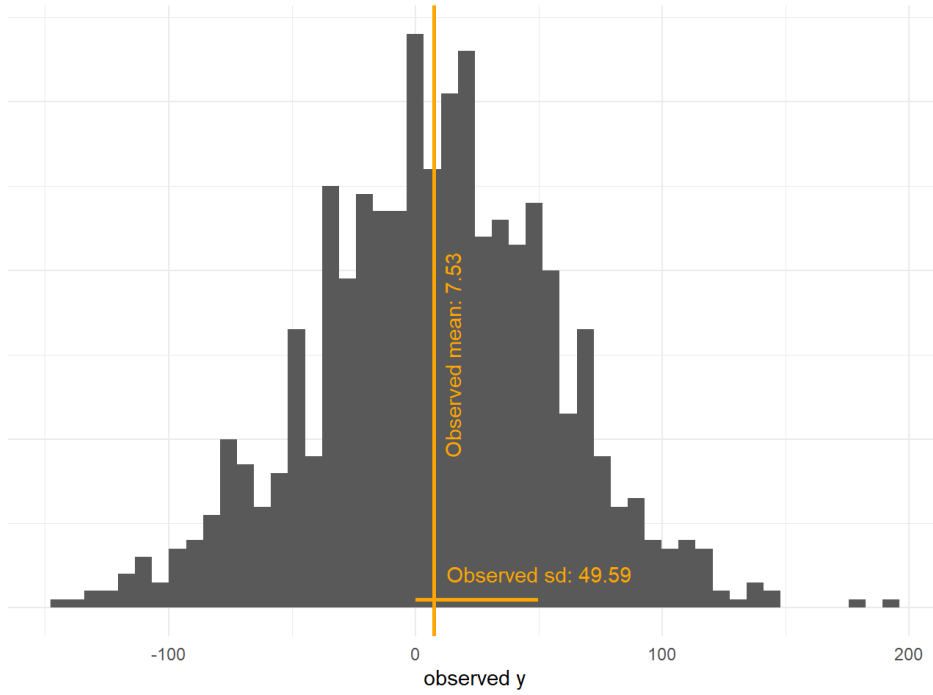


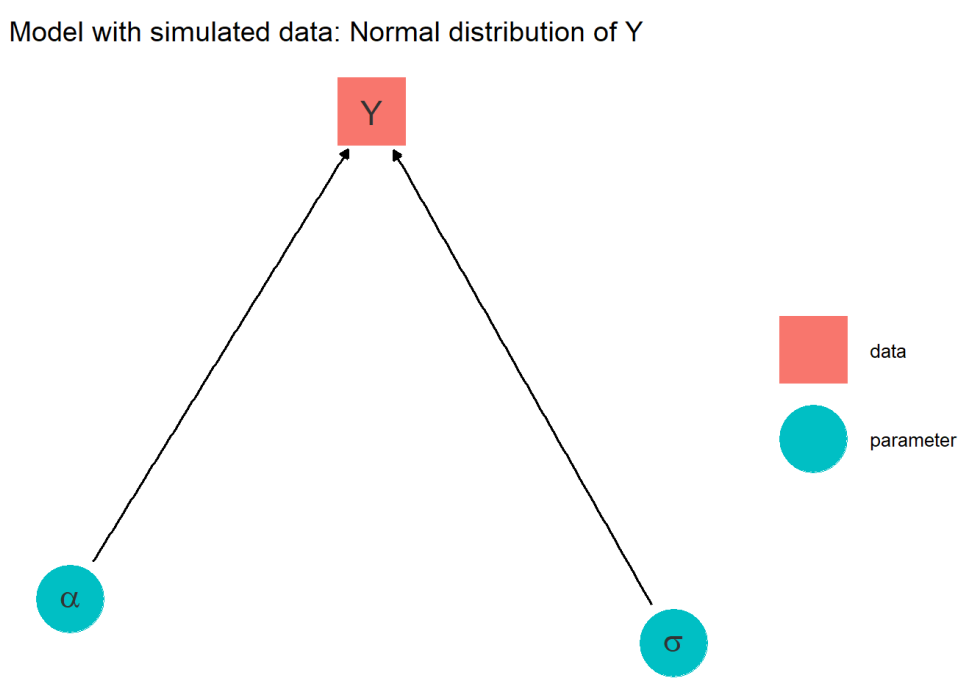
Figure 1: Simulated observations distribution

Modelling the data

Now we want to model y, our observations. We see that y has a clear bell shape. It can thus be **approximate with a Normal distribution** that is:

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

The corresponding DAG is:



We need to give some prior to the two parameters μ the mean and σ the standard deviation. Let's imagine that we don't have access to more information on the variable Y than its manifestation in our dataset.

We can guess that μ is around zero but without great confidence (see Figure other words, we can assume that μ can be drawn from a Normal(0,100).

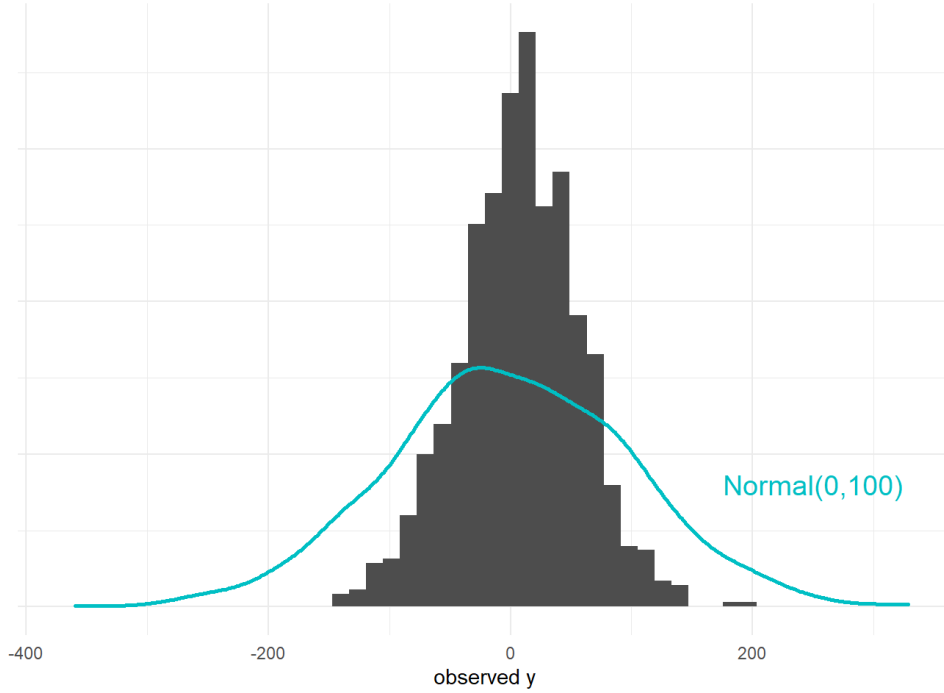


Figure 2: Mu prior distribution

Figure 2 shows what our prior for μ means: we think that μ is likely to be around zero but could be up to 200. Given that our observed y, $\hat{y}$, has no occurrences above 200, it is unlikely that the mean of y is 200 but we don't exclude the possibilty. However we do consider that 0 is more likely than 200.

For σ, we have stricter expectations. Indeed standard deviations are positive. We thus choose as prior for σ a Uniform(0, 200).

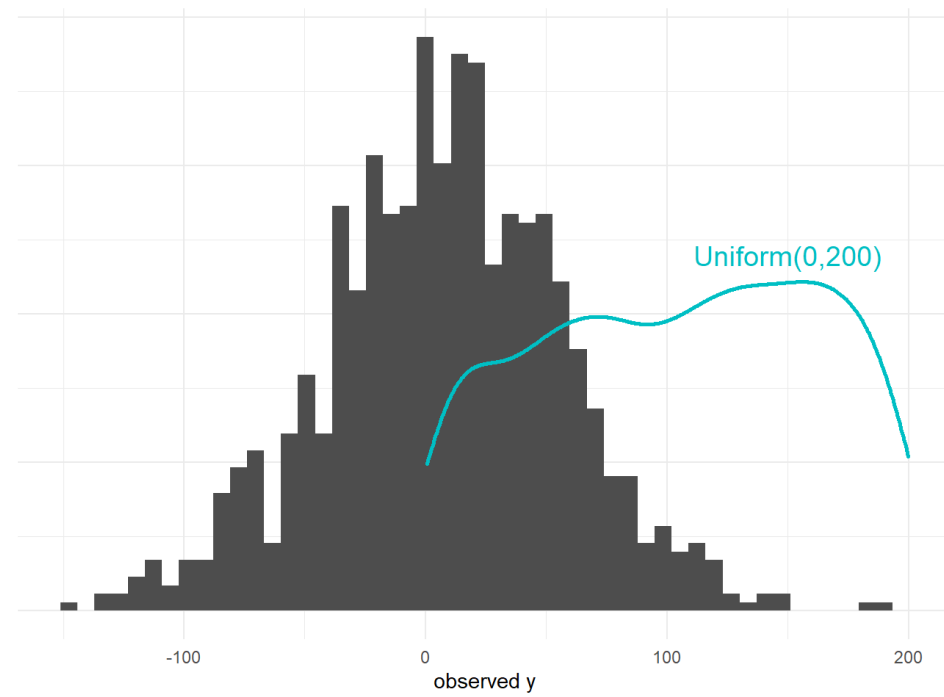


Figure 3: Sigma prior distribution

Figure 3 shows that a uniform prior means a equal probability of σ to be 0 as to be 200.

The final model is:

$$Y \sim \mathrm{Normal}(\mu, \sigma) \tag{2}$$

$$\mu \sim \text{Normal}(0, 100)$$
$$\sigma \sim \text{Uniform}(0, 200)$$

Implementing the model in `stan`

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

```
// Model for simulated data: y as normal distribution
data {
  int<lower=0> n; // number of observations
  vector[n] y; // observations
}

// The parameters accepted by the model. Our model
// accepts two parameters 'mu' and 'sigma'.
parameters {
  real mu;
  real<lower=0> sigma;
}

// The model to be estimated. We model the output
// 'y' to be normally distributed with mean 'mu'
// and standard deviation 'sigma'.
model {
  y ~ normal(mu, sigma);

  mu ~ normal(0,100);
  sigma ~ uniform(0,200);
}
```

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 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_model.stan` in the `tutorial1` folder.

Preparing the data for stan

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

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

Running the model

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

```
# mcmc settings
chains <- 4
warmup <- 250
```

```
iter <- 500
```

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.

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

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

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

```
# mcmc
fit <- rstan::stan(file = file.path('tutorial1_model.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, μ 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.

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

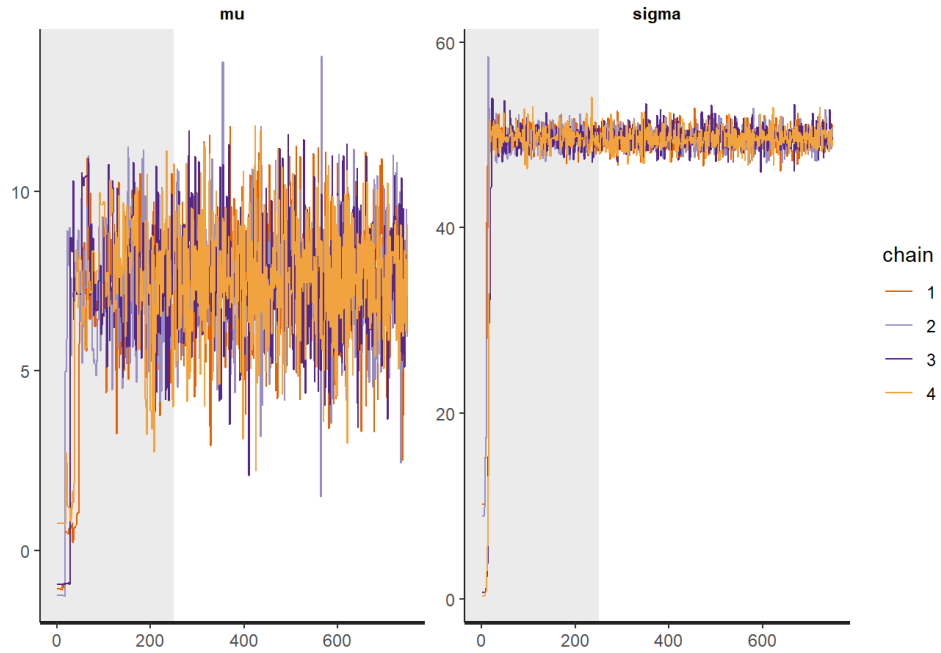


Figure 4: Model Traceplot

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, because the four chains mixed well.

From Figure 4 (and the absence of warning from [redacted]), we can conclude that the model converged.

Evaluating the estimated parameters

Bayesian statistics consider parameters as stochastic, thus it estimates a distribution for each parameter. In practice [redacted] stores parameters estimates for each iteration post-warmup for each chain which gives us 500x4 estimates of the *posterior distribution*.

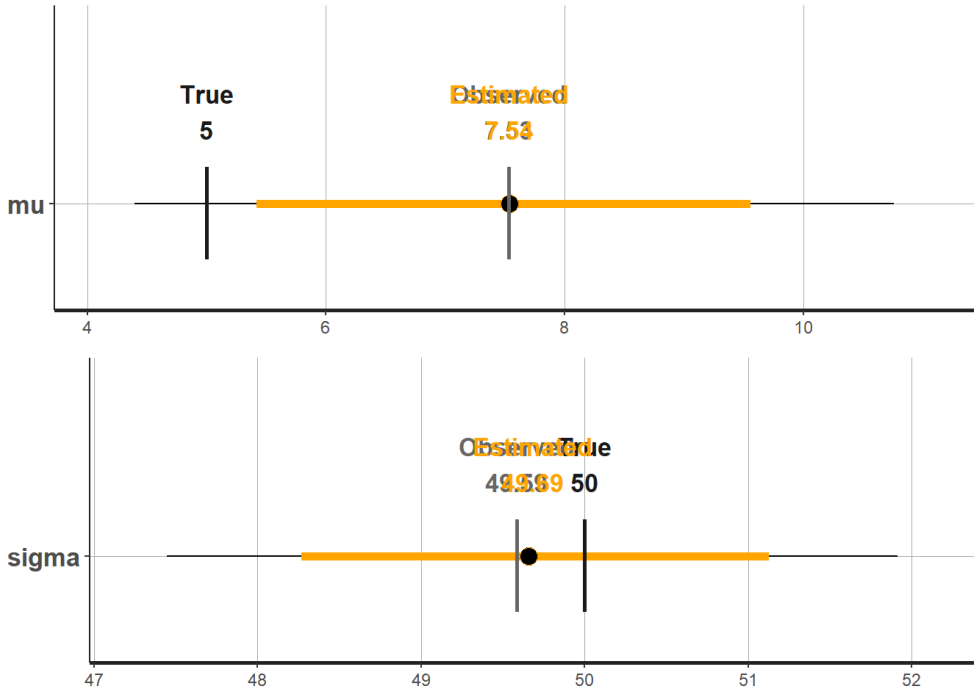
We can extract from the [redacted] object a summary of the estimated distribution using the [redacted] function:

```
estimated <- summary(fit, pars=pars)$summary

estimated %>% kbl() %>% kable_minimal()
```

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
mu	7.538576	0.0429317	1.615445	4.395117	6.471525	7.538612	8.600434	10.75700	1415.884	1.001074
sigma	49.685819	0.0280643	1.131912	47.445472	48.895331	49.656773	50.447527	51.91045	1626.738	1.001501

We can then compare **the parameters** $\hat{\mu}$ **and** $\hat{\sigma}$ with the observed average and standard deviation of $\hat{y}$ as well as the true value, using the [redacted] function.



We see that (1) the observed mean and standard deviation are within the 95% credible intervals of the estimated parameters, (2) the true mean and standard deviation are within the 95% credible intervals of the estimated parameters,

The model structure is inline with the observed data and manages to approximate the true data generating process.

Let’s try with real data

Let’s download the data we will be modelling. It belongs to the supplementary material of the seminal paper describing WorldPop bottom-up population models (Leasure et al. 2020).

```
# 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 of 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 Leasure et al. (2020) and Weber et al. (2018). Survey sites locations are shown in Figure 5.

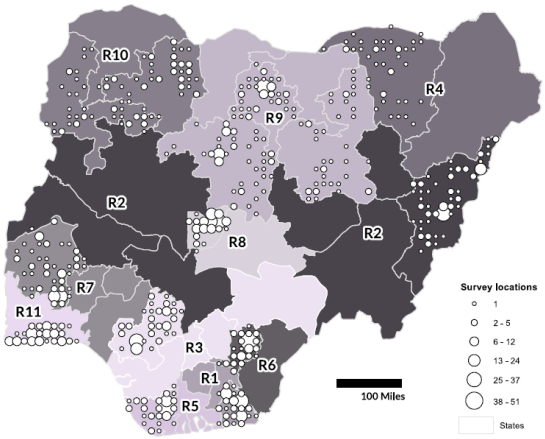


Figure 5: Map of microcensus survey as the number of survey locations within a 20 km grid cell. Region groupings are shaded and numbered R1 - R11. Source: Leasure et al. (2020).

The map in Figure 5 shows some key characteristics of the stratified random 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:

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 ([redacted]) and the settled area ([redacted]) in hectares.

Response variable: the population count

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

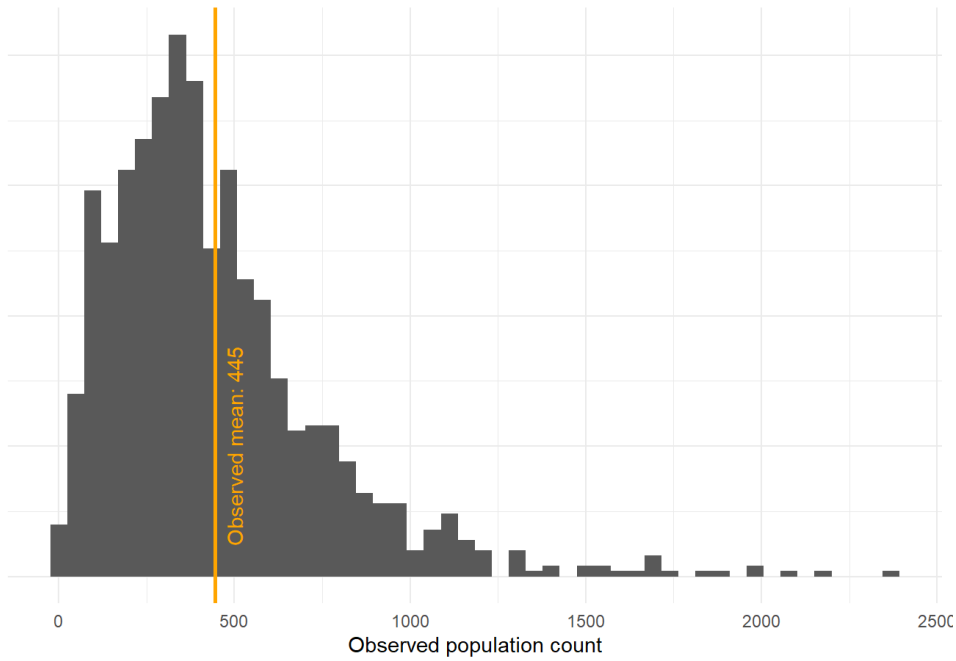


Figure 6: Observed population count distribution at survey sites

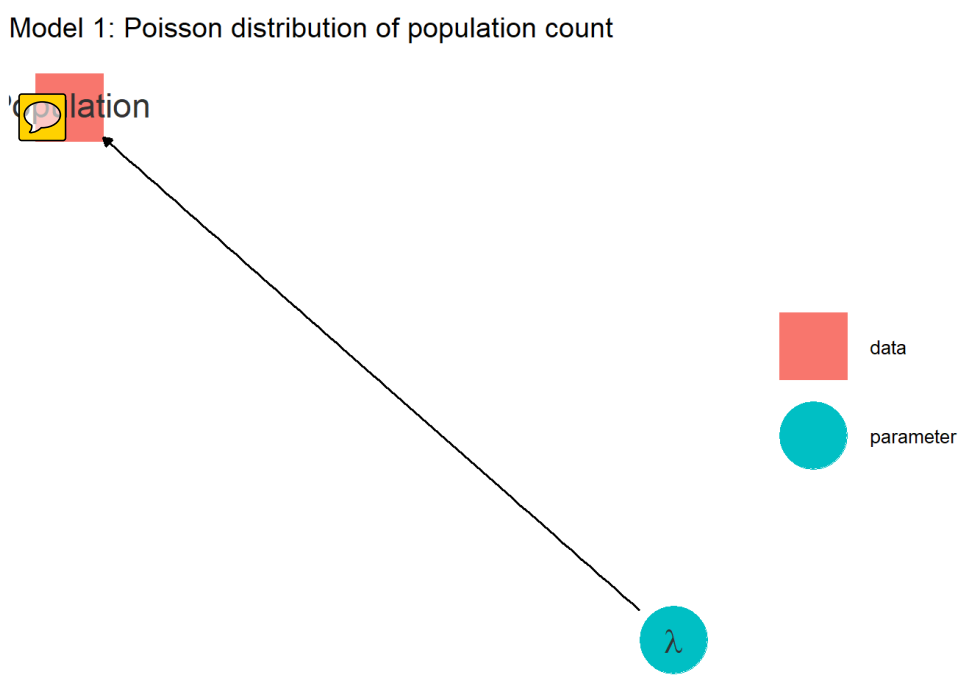
Note the wide variation in population count per survey site, with a maximum of 2370 people.

Modelling Population count: a Poisson distribution

Population count is by definition a discrete, positive variable. A good distribution candidate is the Poisson distribution.

$$\text{population} \sim \text{Poisson}(\lambda)$$

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



We then have to define the prior for λ, which corresponds to the mean of the Poisson distribution. We will choose a relatively uninformed prior based on our understanding of the data.

We know that the mean observed population count is around 450 per cluster. We set up the prior for λ to follow a Uniform between 0 and 3000 to ensure a positive parameter while reducing the information given to the estimation process, and thus the bias introduced.

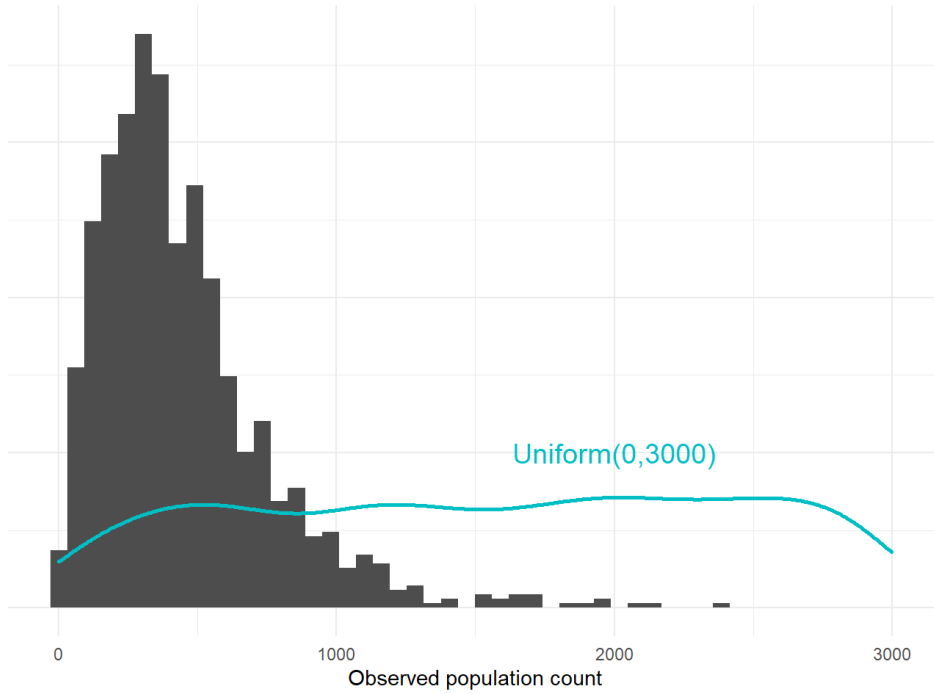


Figure 7: Lambda prior distribution

The final model is:

$$\begin{aligned} \text{population} &\sim \text{Poisson}(\lambda) \\ \lambda &\sim \text{Uniform}(0, 3000) \end{aligned} \tag{3}$$

Implementing the model

To estimate λ, we will write **our first population model** in stan.

```
// Model 1: Population count as a Poisson distribution
data{
  int<lower=0> n; // number of microcensus clusters
  int<lower=0> population[n]; // count of people
}
parameters{
  // rate
  real<lower=0> lambda;
}

model{
  // population totals
  population ~ poisson(lambda);
  // rate
  lambda ~ uniform(0, 3000);
}
```

We declare the input variable [redacted] as integer because our population data are counts and set it up to be positive. We define λ as a positive real.

We store this model under [redacted].

Estimating the model

We prepare the data for [redacted]:

```
# prepare data for stan
```

```
stan_data <- list(
  population = data$N,
  n = nrow(data))
```

We keep the same parameters as previously for the Markov Chain algorithm and declare λ as the parameter to monitor

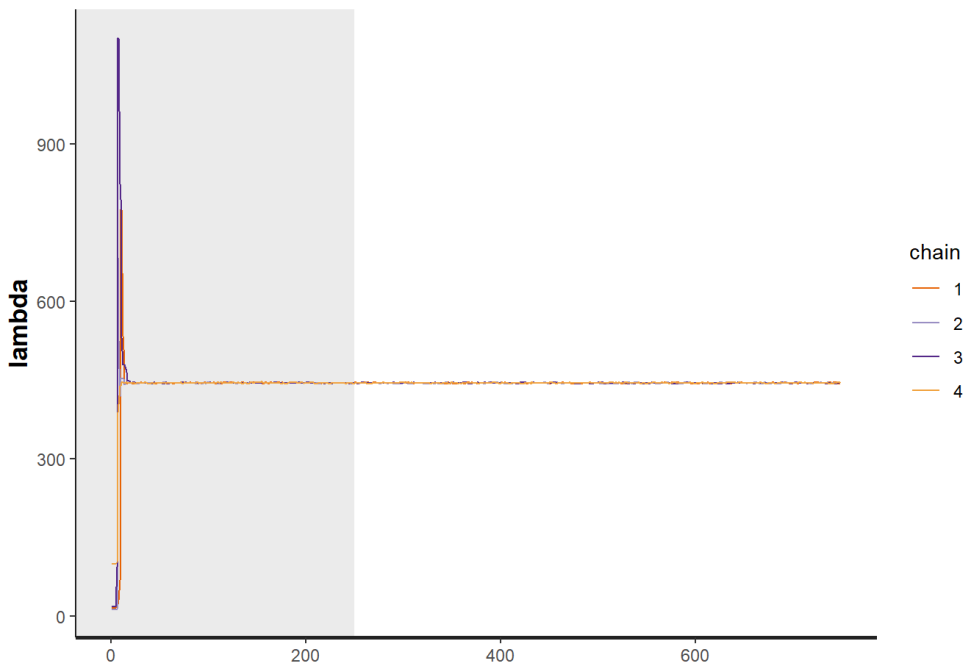
```
# parameters to monitor
pars <- c('lambda')
```

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

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

The traceplot shows a model that converges to the observed mean:

```
traceplot(fit, inc_warmup=T)
```

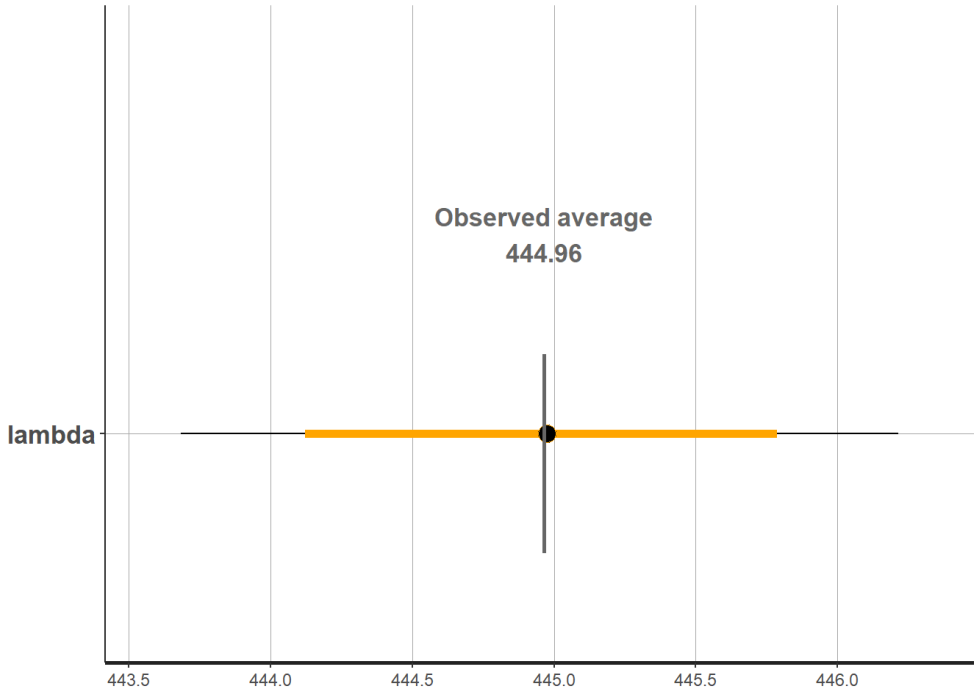


Note the wide variation at the start of the estimation. This is due to the uniform prior for λ declaring that any value between 0 and 3000 is similarly possible.

Evaluating the model goodness-of-fit

Estimated parameters

We plot **the estimated parameter** $\hat{\lambda}$ to see how it compares with the observed average population count.



The estimated mean corresponds to the observed mean.

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** .

```
// Model 2bis: Population count as a normal distribution with integrated predictions
...
generated quantities{
   real population_hat[n];

   for(idx in 1:n){
     population_hat[idx] = poisson_rng( lambda );
   }
}
```

We define the parameter population_hat as a draw (as represented by the suffix rng for *random number generator*) from a Poisson distribution with the estimated $\hat{\lambda}$ for each iteration.

We run the model stored under tutorial1_model1bis.stan:

```
pars <- c('lambda', '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.

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

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

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

iteration	cluster_1	cluster_2	cluster_3	cluster_4	cluster_5	cluster_6	cluster_7	cluster_8	cluster_9	cluster_10
iter_1	430	467	440	421	419	424	457	423	429	416
iter_2	412	443	421	414	454	438	458	418	422	496
iter_3	443	437	422	448	461	461	401	436	430	427
iter_4	458	442	418	448	415	448	454	423	433	439
iter_5	443	421	455	472	437	469	457	417	434	441
iter_6	471	438	482	440	454	476	435	437	472	427
iter_7	448	454	444	450	438	420	395	421	415	438
iter_8	404	434	460	419	453	473	457	448	435	443
iter_9	415	482	441	415	449	408	449	420	424	448
iter_10	449	446	472	444	441	448	493	443	477	421

We get thus a **posterior prediction distribution of population count** for every survey site. Figure 8 shows a posterior distribution for the first survey site.

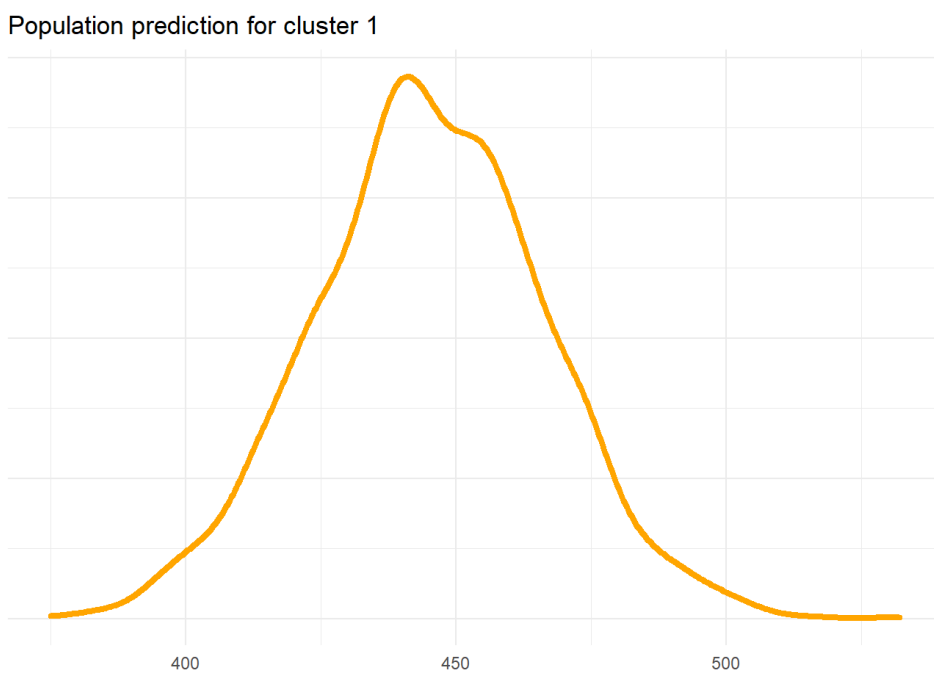


Figure 8: Example of posterior prediction distribution for one cluster

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

```
# 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	445.1085	488.000	402.000
cluster_10	445.0675	488.000	404.000
cluster_100	445.1840	487.025	404.975
cluster_1000	445.6165	484.000	403.000

cluster_1001	445.2855	487.000	404.975
cluster_1002	444.4365	486.000	403.000

We note that all predictions are very similar. The model includes only two parameters — mean and standard deviation — and the site-level predictions do not account for any site-level characteristics. Therefore, predictions at each site are drawn from the exact same distribution.

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.

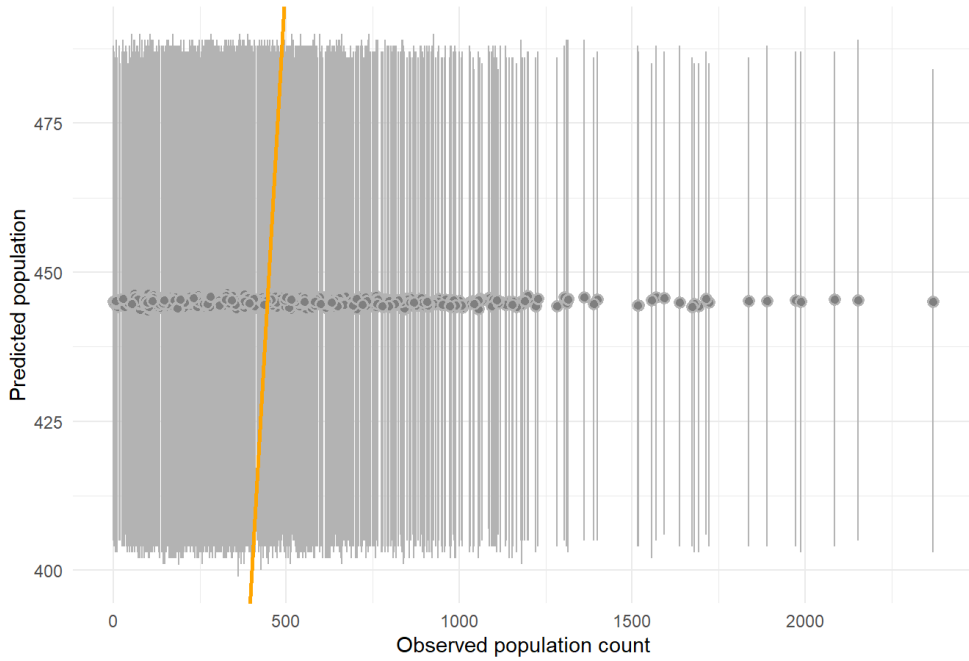


Figure 9: Comparison between observed and predicted population count (with the Poisson model). Orange line indicates the 1:1 line

Figure 9 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), there is no subnational variations introduced. Furthermore the credible intervals entail few observations: few grey lines intersect the orange line. This indicates issues with the modelling.

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 **inaccuracy,** 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

```
# 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 = "Poisson model goodness-of-fit metrics") %>% kable_minimal()
```

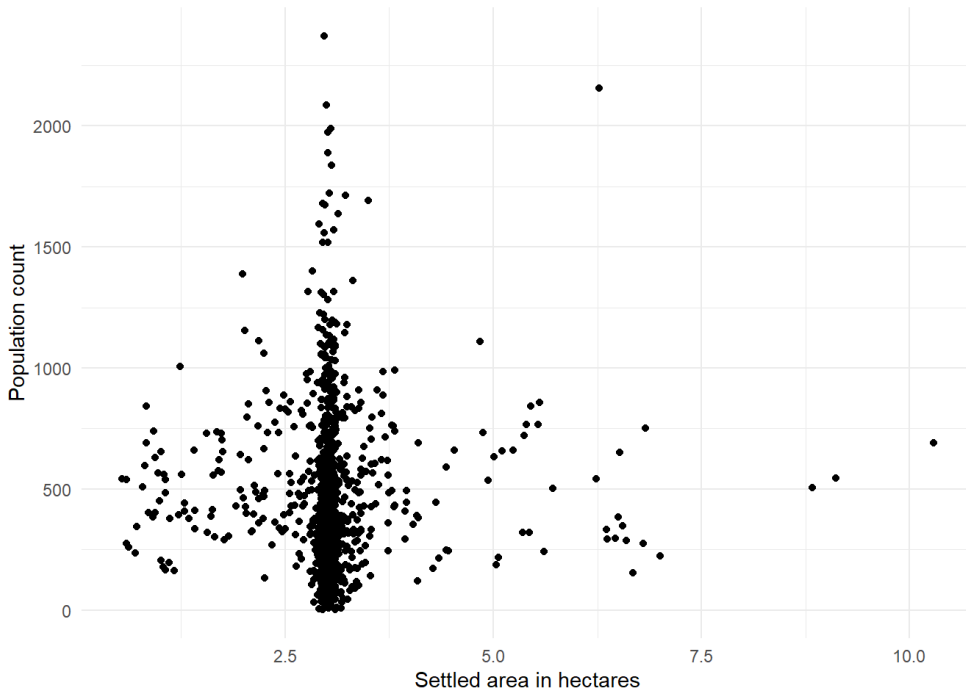
Table 1: Poisson model goodness-of-fit metrics

Bias	Imprecision	Inaccuracy	Correct credible interval (in %)	R2
0.0293225	315.8384	230.9604	10	3.27e-05

Table 1 confirms that the model is incorrectly specified: only 10% of the observations falls in their credible intervals.

This limitation is due to the impossibility to model overdispersion within a Poisson framework. Indeed λ defines both the mean and the variance of a Poisson variable.

A source of overdispersion comes from the size of the clusters. We observed population counts for units with different area, in particular different sizes of settled areas as shown in Figure 1.



Modelling Population count: a Poisson Lognormal model

To **incorporate overdispersion** in our model, we decompose population count as follows:

$$\text{population} = \text{pop\_density} \quad \text{settled\_area}$$

The left hand side of the equation is a discrete observed variable whereas the right hand side is composed of two continuous variables, one observed, the *settled_area* and one latent, the *pop_density*.

Formally we can rewrite Equation (3) as:

$$\text{population} \quad \text{Poisson}(\text{pop\_density} \quad \text{settled\_area})$$

This formulation models a continuous positive latent variable, *pop_density* that can be modelled with its own distribution. We opt for a **Lognormal which is a continuous probability distribution of a random variable whose logarithm is normally distributed**. It is characterised by a positive distribution skewed to the right. The lognormal has two parameters, μ, its location parameter that defines its median and σ its scale parameter that defines its geometric standard deviation. This model allows us to capture overdispersion through σ.

Writing the model

Applied to our population modelling it gives us:

$$\text{population} \sim \text{Poisson}(\text{pop\_density} \quad \text{settled\_area})$$
$$\text{pop\_density} \sim \text{Lognormal}(\mu, \sigma)$$

Note that this equation is equivalent to:

$$\log(\text{pop\_density}) \sim \text{Normal}(\mu, \sigma)$$

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

$$\text{pop\_density} \sim \exp(\text{Normal}(\mu, \sigma))$$

Defining the priors

In the Lognormal, there is two parameters, μ that 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 median of the population density is 4.82 and the observed log geometric standard deviation is 0.87.

We choose as prior for μ a Normal(5,4):

```
## Warning: Removed 2 rows containing non-finite values (stat_density).
```

```
## Warning: Removed 2 rows containing missing values (geom_bar).
```

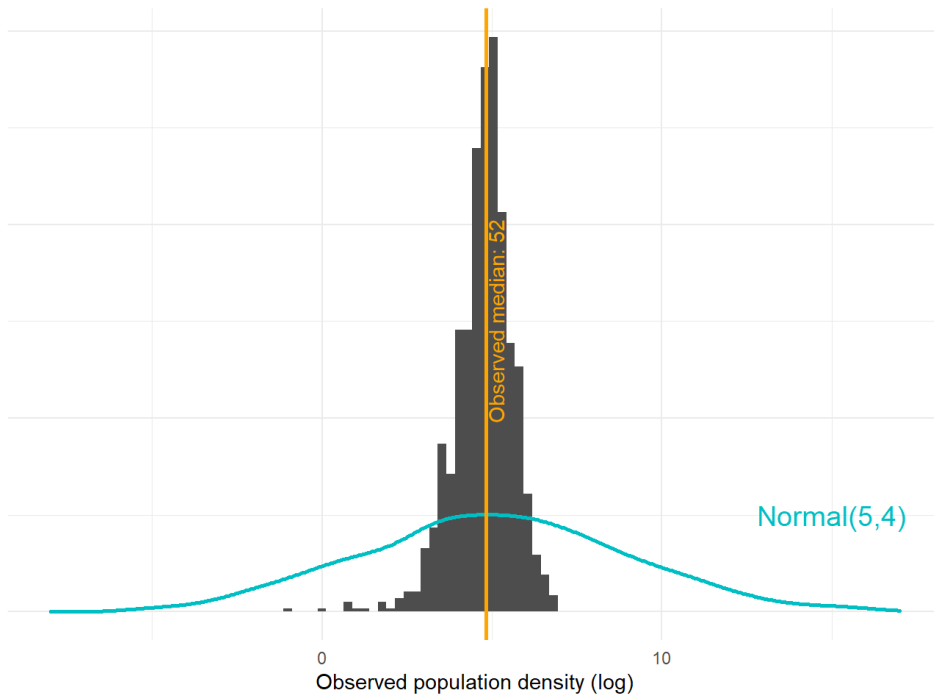


Figure 10: Prior distribution for mu

We choose as prior for σ a Uniform(0,4)

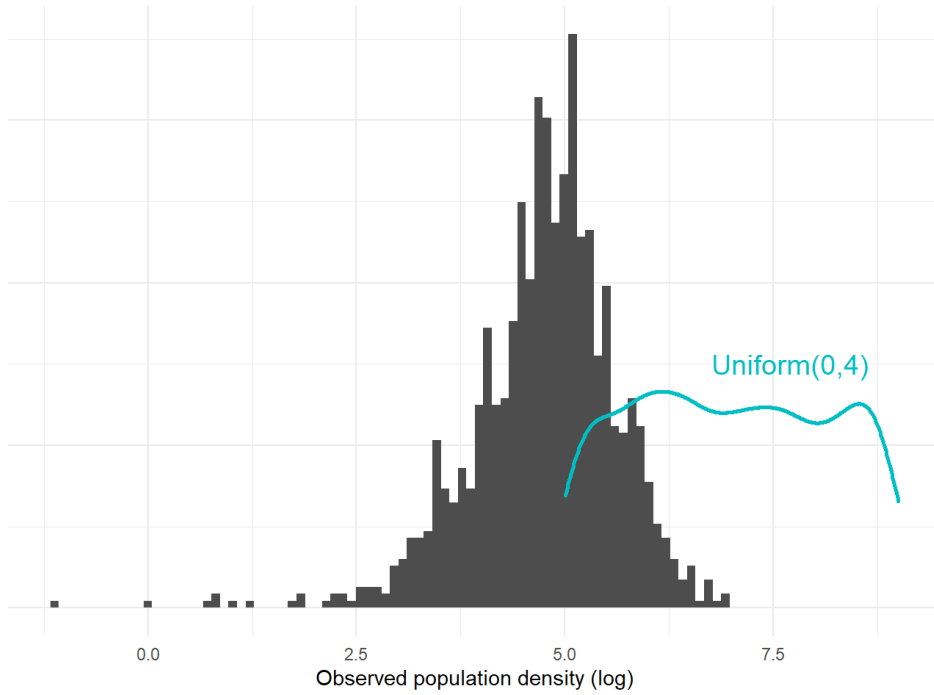


Figure 11: Prior distribution for mu

The resulting model can be written as follows:

$$\begin{aligned} \text{population} &\sim \text{Poisson}(\text{pop\_density} \quad \text{settled\_area}) \\ \text{pop\_density} &\sim \text{Lognormal}(\mu, \sigma) \\ \mu &\sim \text{Normal}(5, 4) \\ \sigma &\sim \text{Uniform}(0, 4) \end{aligned} \tag{4}$$

Implementing the model

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

```
// 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 mu;
  // variance
  real<lower=0> sigma;
}
model{
  // population totals
  population ~ poisson(pop_density .* area);
  pop_density ~ lognormal( mu, sigma );
  // intercept
  alpha ~ normal(5, 4);
  // variance
  sigma ~ uniform(0, 4);
}
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 [redacted], and prepare the corresponding data:

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

Note that the population density is not an input data of the model, but an unobserved latent variable that we model through the Lognormal.

Then we declare the parameters to monitor (including [redacted]) and run the model.

```
# set parameters to monitor
pars <- c('mu','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)
```

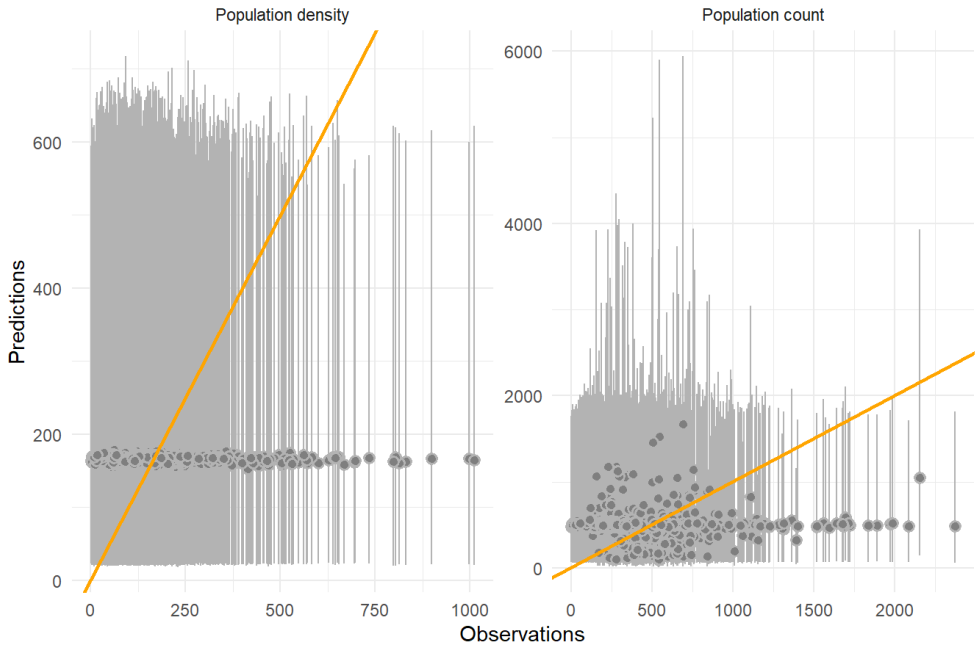
No warnings are shown.

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:

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



We see for the [redacted] the same estimation pattern than in the Poisson model, that is a similar mean posterior prediction distribution for every survey sites. The predicted [redacted] is, in contrast, influenced by the [redacted] and by the variance term that adapts the confidence intervals for each cluster.

We use the same metrics based on residuals to assess the goodness-of-fit of the model.

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

Bias	Imprecision	Inaccuracy	Correct credible interval (in %)	R2
61.51073	338.2973	265.7858	95.2	7.67e-05

The proportion of observations that falls in their credible intervals shows a well-behaved model: 95.5% of the observations falls in the 95% credible interval. In average the model overestimates the population count by around 60 people, but this statistics varies in average by 337 people.

No variations is included in this model: we build a national model of population count. 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.

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

Acknowledgements

This tutorial was written by Edith Darin from WorldPop, University of Southampton and Douglas Leasure from Leverhulme Centre for Demographic Science, University of Oxford.

Funding for the work was provided by the United Nations Population Fund.

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. 2020. “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.

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.

---

1. More details here.