Bayesian pharmacometrics modeling using Stan and Torsten, Part I

Abstract

Stan is an open-source probabilistic programing language, primarily designed to do Bayesian data analysis. Its main inference algorithm is an adaptive Hamiltonian Monte Carlo sampler, supported by state of the art gradient computation. Stan bolsters several strengths, notably efficient computation, an expressive language which offers a great deal of flexibility, and numerous diagnostics that allow modelers to check whether the inference is reliable. Torsten extends Stan with a suite of functions that facilitate the specification of pharmacokinetic and pharmacodynamic models, and makes it straightforward to specify a clinical event schedule. Part I of this tutorial demonstrates how to build, fit, and criticize simple pharmacokinetic models using Stan and Torsten.

1 Introduction

Bayesian inference offers a principled approach to learn about unknown variables from data using a probabilistic analysis. The conclusions we draw are based on the posterior distribution which, in all but the simplest cases, is intractable. We can however probe the posterior using a host of techniques such as Markov chains Monte Carlo sampling and approximate Bayesian computation. Writing these algorithms is a tedious and error prone endeavor but fortunately modelers can often rely on existing software with efficient implementations.

In the field of pharmacometrics, statistical software such as NONMEM [**?**] and Monolix [**?**] support many routines to specify and analyze pharmacokinetic and pharmacodynamic population models. There also exist more general probabilistic languages such as, to only name a few, BUGS [**?**] and more recently Stan [1], which will be the focus of this tutorial. Stan supports a rich library of probability densities, matrix operations and numerical solvers for differential equations. These features make for a rich and flexible language, however writing common pharmacometrics models can be tedious. Torsten extends Stan by providing a suite of functions to facilitate the specification of pharmacometrics models. These functions make it straightforward to model the event schedule of a clinical trial and parallelize computation across patients.

1.1 Why Stan?

We believe that Stan, coupled with Torsten, can be an important addition to the pharmacometrician's toolkit, especially for Bayesian data analysis.

The most obvious strength of Stan is its flexibility: it is straightforward to specify priors, systems of ODEs, a broad range of measurement models, missing data models and hierarchies. Because of this flexibility, various data sources and their corresponding measurement models can be combined into one large model, over which full Bayesian inference can be performed [e.g **?**].

Secondly, Stan offers efficient inference algorithms, most notably its adaptive *Hamiltonian Monte Carlo* sampler, a gradient-based Markov chains Monte Carlo algorithm [**?** **?**], automatic differentiation variational inference [**?**], and penalized maximum likelihood estimators. Stan's inference algorithms are supported by a state-of-the-art automatic differentiation library, which efficiently generates the requisite derivatives [**?**].

Thirdly, Stan runs many diagnostics – including the detection of divergent transitions [**?**], and the improved computation of effective sample sizes and scale reduction factors, $\hat{R}$ [**?**], and more – and gives detailed warning messages. This makes it considerably easier to identify issues with our inference and our models. Several of these tools improve commonly used diagnostics which may not detect important problems, in which case our inference fails without us realizing it. Stan fails better: it fails loudly.

Last but not least, both Stan and Torsten are open-source projects, meaning they are free and their source code can be examined and, if needed, scrutinized. The projects are under active development with new features being added regularly.

1.2 Bayesian inference: notation, goals, and comments

Given observed data, $\mathcal{D}$, and latent variables, θ, a Bayesian model is defined by the joint distribution, $p(\mathcal{D}, \theta)$. The latent variables can include model parameters, missing data, and more. In this tutorial, we are mostly concerned with estimating model parameters.

The joint distribution observes a convenient decomposition,

$$p(\mathcal{D}, \theta) = p(\theta)p(\mathcal{D} \mid \theta),$$

with $p(\theta)$ the *prior* distribution and $p(\mathcal{D} \mid \theta)$ the *likelihood*. The prior encodes information about the parameters, usually based on scientific expertise or results from previous analysis. The likelihood tells us how the data is distributed for a fixed parameter value and, per one interpretation, can be thought of as a "story of how the data is generated" [**?**]. The Bayesian proposition is to base our inference on the *posterior* distribution, $p(\theta \mid \mathcal{D})$.

It is worth pointing out that the posterior density is an unfathomable object which lives in a high dimensional space. There is no such thing as "computing the posterior distribution". We cannot even numerically evaluate the posterior density at any particular point! Instead we must probe the posterior distribution and learn the characteristics that interest us the most. In our experience, this often includes a measure of a central tendency and a quantification of uncertainty, for example the mean and the variance, or the median and the 5^{th} and 95^{th} quantiles. For skewed or multimodal distributions, we may want a more refine analysis which looks at many quantiles. What we compute are estimates of these

quantities. One strategy is to generate *approximate* samples from the posterior distribution and then use sample mean, sample variance, and sample quantiles as our estimators.

Bayes' rule teaches us that

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{p(\mathcal{D})}.$$

Typically we can evaluate the joint density in the nominator but not the normalizing constant in the denominator. A useful method must therefore be able to generate samples using the *unnormalized* posterior density. Many Markov chains Monte Carlo (MCMC) algorithms are designed to do exactly this. Starting at an initial point, these chains explore the parameter space, Θ, one iteration at a time, to produce the desired samples. Hamiltonian Monte Carlo (HMC) is an MCMC method which uses the gradient to efficiently move across the parameter space [**?**]. Computationally, running HMC requires evaluating $\log p(\mathcal{D}, \theta)$ and $\nabla_\theta \log p(\mathcal{D}, \theta)$ many times across Θ, i.e. for varying values of θ but fixed values of $\mathcal{D}$. For this procedure to be well-defined, θ must be a continuous variable, else the requisite gradient does not exist. Discrete parameters require a special treatment, which we will not discuss in this tutorial.

A Stan program specifies a method to evaluate $\log p(\mathcal{D}, \theta)$. Thanks to automatic differentiation, this implicitly defines a procedure to compute $\nabla_\theta \log p(\mathcal{D}, \theta)$. Together, these two objects provide all the relevant information about our model to run HMC sampling and other gradient-based inference algorithms.

1.3 Bayesian workflow

Bayesian inference is only one step of a broader modeling process, which we might call the Bayesian workflow [**?** **?**]. Once we fit the model, we need to check the inference and if needed, fine tune our algorithm, or potentially change method. And once we trust the inference, we naturally need to check the fitted model. Our goal is to understand the shortcomings of our model and motivate useful revisions. During the early stages of model development, this mostly comes down to troubleshooting our implementation and later this "criticism" step can lead to deeper insights.

All through the tutorial, we will demonstrate how Stan and Torsten can be used to check our inference and our fitted model.

1.4 Setting up Stan and Torsten

Detailed instructions on installing Stan and Torsten can be found on `https://github.com/metrumresearchgroup/Torsten`. At its core, Stan is a C++ library but it can be interfaced with one of many scripting languages, including R, Python, and Julia. We will use cmdStanR, which is a lightweight wrapper of Stan in R, and in addition, the packages Posterior [**?**] BayesPlot [**?**], and Loo [**?**].

The R and Stan code for all the examples are available at `link`.

1.5 Prerequisites and resources

Our aim is to provide a self-contained discussion, while trying to remain concise. We assume the reader is familiar with compartment models as they arise in pharmacokinetic and pharmacodynamic models, and has experience with data that describe a clinical event schedule. For the latter, we follow the convention established by NONMEM. Exposure to Bayesian statistics and inference algorithms is desirable, in particular an elementary understanding of standard MCMC methods. We expect the reader to know R but we don't assume any background in Stan.

Helpful reads include the *Stan User Manual* [**?**] and the *Torsten User Manual* [**?**]. A comprehensive textbook on Bayesian modeling is *Bayesian Data Analysis* by Gelman et al (2013) [**?**], with more recent insights on the Bayesian workflow provided by Gelman et al (2020) [**?**]. Betancourt (2018) offers an accessible discussion on MCMC methods, with an emphasis on HMC [**?**]. We will discuss in the conclusion additional readings which can complement this tutorial.

2 Two compartment model

As a starting example, we consider a compartment pharmacokinetic model for a single patient. The patient receives multiple doses at regular time intervals and the drug plasma concentration is recorded over time. Our goal is to infer the physiological parameters of the patient, pertinent to the drug's pharmacokinetics, and the measurement error.

2.1 Pharmacokinetic model and clinical event schedule

The two compartment pharmacokinetic model describes how the drug circulates in the patient's body, until it is cleared out (Figure 1). The drug is orally administered and enters the system through the gut. Once the drug is introduced in the system, the *natural evolution* of the system is described by a system of ODEs. In the case of a two compartment model with a first-order absorption from the gut, the system is the following:

$$\begin{aligned}
\frac{\mathrm{d}y_{\text{gut}}}{\mathrm{d}t} &= -k_a y_{\text{gut}} \\
\frac{\mathrm{d}y_{\text{central}}}{\mathrm{d}t} &= k_a y_{\text{gut}} - \left(\frac{CL}{V_{\text{cent}}} + \frac{Q}{V_{\text{cent}}}\right) y_{\text{cent}} + \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \\
\frac{\mathrm{d}y_{\text{peri}}}{\mathrm{d}t} &= \frac{Q}{V_{\text{cent}}} y_{\text{cent}} - \frac{Q}{V_{\text{peri}}} y_{\text{peri}}
\end{aligned}$$

with

- $y(t)$: the drug mass in each compartment (mg),
- k_a: the rate constant at which the drug flows from the gut to the central compartment (h^{-1}),
- Q: the clearance at which the drug flows back and forth between the central and the peripheral compartment (L/h),

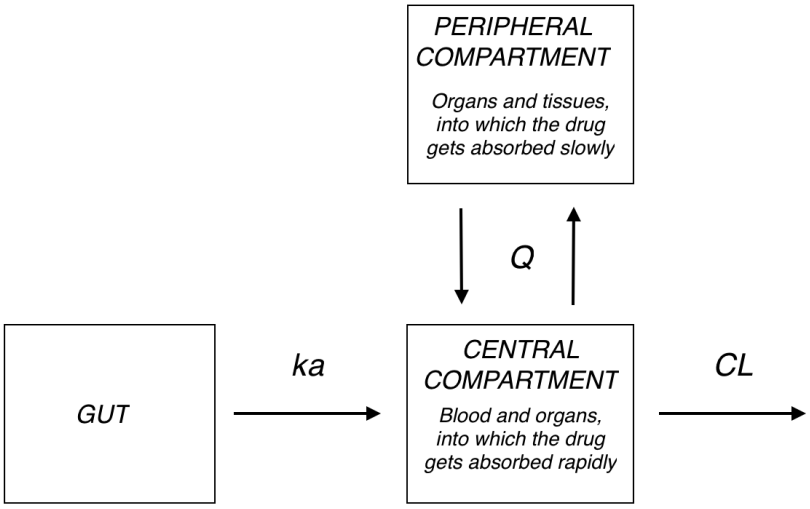

Figure 1: Two compartment model with first-order absorption from the gut.

- CL: the clearance at which the drug is cleared from the central compartment (L / h),
- V_{cent}: the volume of the central compartment (L),
- V_{peri}: the volume of the peripheral compartment (L).

During the trial, the patient receives a dose of 1,200 mg every 12 hours, until they have received a total of 14 doses. Measurements are taken shortly after the first, second, and last doses, and at regular intervals during the treatment. Both intervention and measurement events are described by the event schedule, which follows the convention established by NONMEM. This means the user must provide for each event the following variables: `cmt`, `evid`, `addl`, `ss`, `amt`, `time`, `rate`, and `ii`. Appendix **??** reviews the role of each variable.

2.2 Statistical model

Given a treatment, x, and the physiological parameters, $\{k_a, Q, CL, V_{\text{cent}}, V_{\text{peri}}\}$, we obtain the exact drug plasma concentration, c, by solving the relevant ODE. Our measurements, y, are a perturbation of c because of our measurement error. We model this error using a lognormal error, that is

$$\log y \sim \text{Normal}(\log c, \sigma^2),$$

where σ is a standard deviation we wish to estimate. The deterministic computation of c along with the measurement model, defines our likelihood function $p(y \mid \theta, x)$, where $\theta = \{k_a, Q, CL, V_{\text{cent}}, V_{\text{peri}}, \sigma\}$.

It remains to define a prior distribution, $p(\theta)$. Our prior should allocate probability mass to every plausible parameter value and exclude patently absurd values. For example the volume of the central compartment is on the order of ten liters, but it cannot be the size of the Sun. In this simulated example, our priors for the individual parameters are based on population estimates from previous (hypothetical) studies.

$$\begin{aligned} CL &\sim \text{logNormal}(\log(10), 0.25); \\ Q &\sim \text{logNormal}(\log(15), 0.5); \\ V_{\text{cent}} &\sim \text{logNormal}(\log(35), 0.25); \\ V_{\text{peri}} &\sim \text{logNormal}(\log(105), 0.5); \\ k_a &\sim \text{logNormal}(\log(2.5), 1); \\ \sigma &\sim \text{Half} - \text{Normal}(0, 1); \end{aligned}$$

Suggestions for building priors can be found in references [**? ? ?**].

2.3 Specifying a model in Stan

We can now specify our statistical model using a Stan file, which is divided into coding blocks, each with a specific role. From R, we then run inference algorithms which take this Stan file as an input.

2.3.1 Data and parameters block

To define a model, we need a procedure which returns the log joint distribution, $\log p(\mathcal{D}, \theta)$. Our first task is to declare the data, $\mathcal{D}$, and the parameters, θ, using the coding blocks `data` and `parameters`. It is important to distinguish the two. The data is fixed. By contrast, the parameter values change as HMC explores the parameter space, and gradients of the joint density are computed with respect to θ, but not $\mathcal{D}$.

For each variable we introduce, we must declare a type and, for containers such as arrays, vectors, matrices, etc., the size of the container. In addition, each statement ends with a semi-colon. It is possible to specify constraints on the parameters, using the keywords `lower` and `upper`. If one of these constraints is violated, Stan returns an error message. More importantly, parameters are transformed into unconstrained parameters – for instance, positive variables are put on the log scale – which greatly improves computation.

```
data {
  int<lower = 1> nEvent;
  int<lower = 1> nObs;
  int<lower = 1> iObs[nObs];  // index of events which
                              // are observations.

  // Event schedule
  int<lower = 1> cmt[nEvent];
  int evid[nEvent];
```

```
  int addl[nEvent];
  int ss[nEvent];
  real amt[nEvent];
  real time[nEvent];
  real rate[nEvent];
  real ii[nEvent];

  // observed drug concentration
  vector<lower = 0>[nObs] cObs;
}

parameters {
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> VC;
  real<lower = 0> VP;
  real<lower = 0> ka;
  real<lower = 0> sigma;
}
```

2.3.2 model block

Next, the `model` block allows us to modify the variable `target`, which Stan recognizes as the log joint distribution. The following statement increments `target` using the prior on σ, which is a normal density, truncated at 0 to only put mass on positive values.

```
  target += normal_lpdf(sigma | 0, 1);
```

The truncation is implied by the fact σ is declared as lower-bounded by 0 in the parameters block. An alternative syntax is the following:

```
  sigma ~ normal(0, 1);
```

This statement now looks like our statistical formulation and makes the code more readable. But we should be mindful that this is not a sampling statement, rather instructions on how to increment `target`. We now give the full model block:

```
model {
  // priors
  CL ~ lognormal(log(10), 0.25);
  Q ~ lognormal(log(15), 0.5);
  VC ~ lognormal(log(35), 0.25);
  VP ~ lognormal(log(105), 0.5);
  ka ~ lognormal(log(2.5), 1);
  sigma ~ normal(0, 1);

  // likelihood
  logCObs ~ normal(log(concentrationObs), sigma);
}
```

The likelihood statement involves terms which we have not defined yet, notably `logCObs` and `concentrationObs`. These variables are transformations of the data and the parameters, and motivate two additional blocks.

2.3.3 Transformed data and transformed parameters block

In `transformed data`, we can construct variables which only depend on the data. For example,

```
  transformed data {
    vector[nObs] logCObs = log(cObs);
    int nCmt = 3;
    int nTheta = 5;
  }
```

We also specify the number of compartments in our model (including the gut), `nCmt`, and the numbers of physiological parameters, `nTheta`, which will come in handy shortly. Because the data is fixed, this operation is only computed once. By contrast, operations in the `transformed parameters` block need to be performed for each new parameter values.

To compute `concentrationObs` we need to solve the relevant ODE within the clinical event schedule. Torsten provides a function which returns the drug mass in each compartment at each time point in the event schedule.

```
  matrix<lower = 0>[nCmt, nEvent] mass =
   pmx_solve_twocpt(time, amt, rate, ii, evid, cmt,
                    addl, ss, theta);
```

The first eight arguments define the event schedule and the last argument, `theta`, is an array containing the physiological parameters, and defined as follows:

```
  real theta[nTheta] = {CL, Q, VC, VP, ka};
```

The Torsten function we have chosen to use solves the ODEs analytically. Other routines use a matrix exponential, a numerical solver, or a combination of methods [**?**]. It now remains to compute the concentration in the central compartment at the relevant times. The full `transformed parameters` block is then:

```
transformed parameters {
  real theta[nTheta] = {CL, Q, VC, VP, ka};
  row_vector<lower = 0>[nEvent] concentration;
  row_vector<lower = 0>[nObs] concentrationObs;
  matrix<lower = 0>[nCmt, nEvent] mass;

  mass = pmx_solve_twocpt(time, amt, rate, ii, evid,
                          cmt, addl, ss, theta);

  concentration = mass[2, ] ./ VC;
  concentrationObs = concentration[iObs];
}
```

The Stan file contains all the coding blocks in the following order: `data`, `transformed data`, `parameters`, `transformed parameters`, `model`. Appendix **??** provides the full Stan code.

2.4 Calling Stan from R

The package CmdStanR allows us to run a number of algorithms on a model defined in a Stan file. An excellent place to get started with the package is `https://mc-stan.org/cmdstanr/articles/cmdstanr.html`.

The first step is to "transpile" the file – call it `twocpt.stan` –, that is translate the file into C++ and then compile it.

```
mod <- cmdstan_model("twocpt.stan")
```

We can then run Stan's HMC sampler by passing in the requisite data and providing other tuning parameters, such as the number of sampling and warmup iterations, and the number of Markov chains we run.

```
fit <- mod$sample(data = data, chains = n_chains,
                  init = init,
                  parallel_chains = n_chains,
                  iter_warmup = 500,
                  iter_sampling = 500)
```

There are several other arguments we can pass to the sampler and which we will take advantage of throughout the tutorial. For applications in pharmacometrics, we recommend specifying the initial starting points via the `init` argument. In this in future examples, we will draw the initial points from their priors, by defining an appropriate R function.

The resulting `fit` object stores the samples generated by HMC from which can deduce the sample mean, sample variance, and sample quantiles of our posterior distribution. This information is readily accessible using `fit$summary()` and summarized in table 1. We could also extract the samples and perform any number of operations on them.

2.5 Checking our inference

Unfortunately there is no guarantee that a particular algorithm will work across all the applications we will encounter. We can however make sure that certain necessary conditions do not break. Much of the MCMC literature focuses on estimating expectation values, and we will use these results to develop some intuition.

2.5.1 Central limit theorem

For any function f of our latent parameters θ, we define the posterior mean to be

$$\mathbb{E}f = \int_{\Theta} f(\theta)p(\theta \mid \mathcal{D})\mathrm{d}\theta,$$

a quantity also termed the *expectation value*. If we were able to generate exact independent samples,

$$\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n)} \overset{\text{i.i.d}}{\sim} p(\theta \mid \mathcal{D}),$$

	mean	**median**	**sd**	**mad**	**q5**	**q95**	$\hat{\mathbf{R}}$	$\mathbf{ESS}_{\text{bulk}}$	$\mathbf{ESS}_{\text{tail}}$
CL	10.0	10.0	0.378	0.367	9.39	10.6	1.00	1580	1348
Q	19.8	19.5	4.00	4.01	13.8	26.8	1.00	985	1235
V_{cent}	41.2	40.8	9.71	9.96	25.6	57.7	1.00	732	1120
V_{peri}	124	123	18.0	18.0	97.1	155	1.00	1877	1279
k_a	1.73	1.67	0.523	0.522	1.01	2.68	1.00	762	1108
σ	0.224	0.222	0.0244	0.0232	0.187	0.269	1.01	1549	1083

Table 1: Summary of results when fitting a two compartment model. *The first columns return sample estimates of the posterior mean, median, standard deviation, median absolute deviation,* 5^{th} *and* 95^{th} *quantiles, based on our approximate samples. The next three columns return the* $\hat{R}$ *statistics and the effective sample size for bulk and tail estimates, and can be used to identify problems with our inference.*

one sensible estimator would be the sample mean,

$$\hat{\mathbb{E}}f = \frac{1}{n}\sum_{i=1}^{n} f\left(\theta^{(i)}\right).$$

Then, provided the variance of f is finite, the *central limit theorem* teaches us that

$$\hat{\mathbb{E}}f \overset{\text{approx}}{\sim} \text{Normal}\left(\mathbb{E}f, \frac{\text{Var}f}{n}\right).$$

This is a powerful result for two reasons: first it tells us that our estimator is unbiased and more importantly that the expected squared error is driven by the variance of f divided by our sample size, n.

Unfortunately, estimates constructed with MCMC samples will, in general, neither be unbiased, nor will their variance decrease at rate n. For our estimators to be useful, we must therefore check that our samples are unbiased and then use a corrected central limit theorem.

2.5.2 Checking for bias with $\hat{R}$

MCMC samples are biased for several reasons. Perhaps the most obvious one is that Markov chains generate correlated samples, meaning any sample has some correlation with the initial point. If we run the algorithm for enough iterations, the correlation to the initial point becomes negligible and the chain "forgets" its starting point. But what constitutes enough iterations? It isn't hard to construct examples where removing the initial bias in any reasonable time is a hopeless endeavor [e.g. **? ?**].

To identify this bias, we run multiple Markov chains, each started at different points, and check that they all convergence to the region of the parameter space. More precisely, we shouldn't be able to distinguish the Markov chains based on

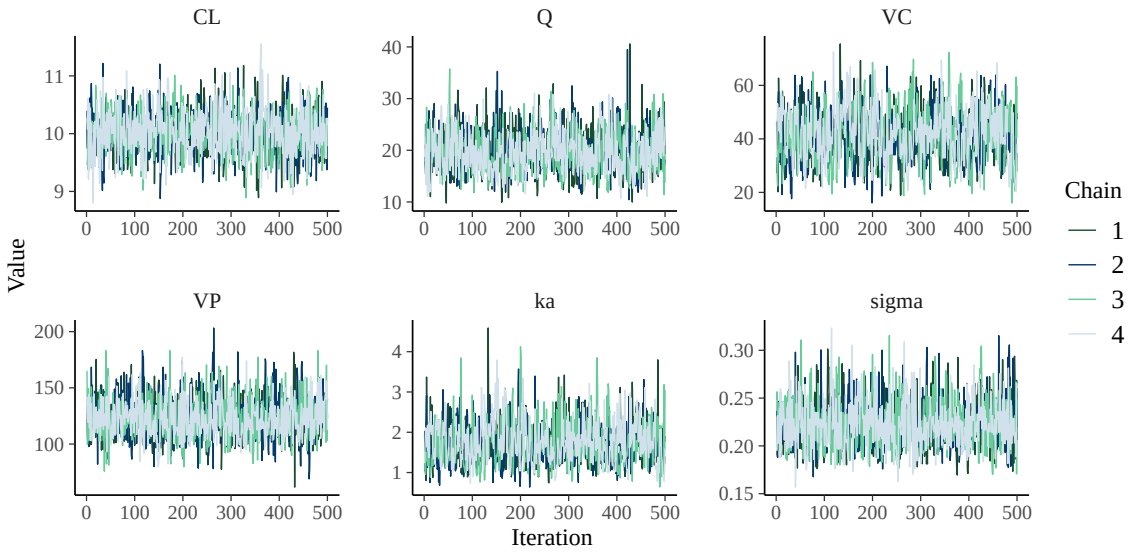

Figure 2: Trace plots. *The sampled values for each parameters are plotted against the iterations during the sampling phase. Multiple Markov chains were initialized at different points. However, once in the sampling phase, we cannot distinguish them.*

the samples alone. One way to check this is to compute the $\hat{R}$ statistics, for which we provide an intuitive definition:

$$\hat{R} \overset{\text{intuitively}}{=} \frac{\text{Between chain variance}}{\text{Within chain variance}}.$$

If the chains are mixing properly, then $\hat{R} \approx 1.0$, as is the case in table 1. Stan uses an improved $\hat{R}$ statistics described in a recent paper by **?**]. We can also visually check that the chains are properly mixing using a trace plot (Figure 2).

If $\hat{R} \gg 1$ and, more generally, if the chains were not mixing, this would be cause for concern and an invitation to adjust our inference method. Even when $\hat{R} = 1$, we should entertain the possibility that all the chains suffer from the same bias. Stan offers additional diagnostics to identify sampling biases, notably by reporting *divergent transitions* of the HMC sampler, a topic we will discuss when we fit more sophisticated models.

2.5.3 Controlling the variance of our estimator

Let's assume that our samples are indeed unbiased. The expected error of our estimator is now determined by the variance. Under certain regularity conditions, our estimator follows an MCMC central limit theorem,

$$\hat{\mathbb{E}}f \overset{\text{approx}}{\sim} \text{Normal}\left(\mathbb{E}f, \frac{\text{Var}f}{n_{\text{eff}}}\right).$$

where the key difference is that $\mathrm{Var} f$ is now divided by the *effective sample size*, n_{eff}, rather than the sample size. This change accounts for the fact our samples are not independent: their correlation induces a loss in information, which increase the variance of our estimator. For CL, we have 2,000 samples, but the effective sample size is 1,580 (Table 1). If n_{eff} is low, our estimator may not be precise enough and we should generate more samples.

The effective sample size is only formally defined in the context of estimators for expectation values. We may also be interested in tail quantities, such as extreme quantiles, which are much more difficult to estimate and require many more samples to achieve a desired precision. **?**] propose a generalization of the effective sample size for such quantities, and introduce the *tail effective sample size*. This is to be distinguished from the traditional effective sample size, henceforth the *bulk effective sample size*. Both quantities are reported by Stan.

2.6 Checking the model: posterior predictive checks

Once we develop enough confidence in our inference, we still want to check our fitted model. There are many ways of doing this. We may look at the posterior distribution of an interpretable parameter and see if it suggests implausible values [e.g **?**]. Or we may evaluate the model's ability to perform a certain task, e.g. classification or prediction, as is often done in machine learning. In practice, we find it useful to do *posterior predictive checks*, that is simulate data from the fitted model and compare the simulation to the observed data [**?** , chapter 6]. Mechanically, the procedure is straightforward:

1. Draw the parameters from their posterior, $\tilde{\theta} \sim p(\theta \mid y)$.
2. Draw new observations from the likelihood, conditional on the drawn parameters, $\tilde{y} \sim p(y \mid \tilde{\theta})$.

This amounts to drawing observations from their posterior distribution, that is $\tilde{y} \sim p(\tilde{y} \mid y)$. The uncertainty due to our estimation and the uncertainty due to our measurement error are then propagated to our predictions.

Stan provides a `generated quantities` block, which allows us to compute values, based on sampled parameters. In our two compartment model example, the following code draws new observations from the likelihood:

```
generated quantities {
  real concentrationObsPred[nObs]
    = exp(normal_rng(log(concentrationObs), sigma));
}
```

Here, we generated predictions at the observed points for each sampled point, $\theta^{(i)}$. This gives us a sample of predictions and we can use the 5^{th} and 95^{th} quantiles to construct a credible interval. We may then plot the observations and the credible intervals (Figure 3) and see that, indeed, the data generated by the model is consistent with the observations.

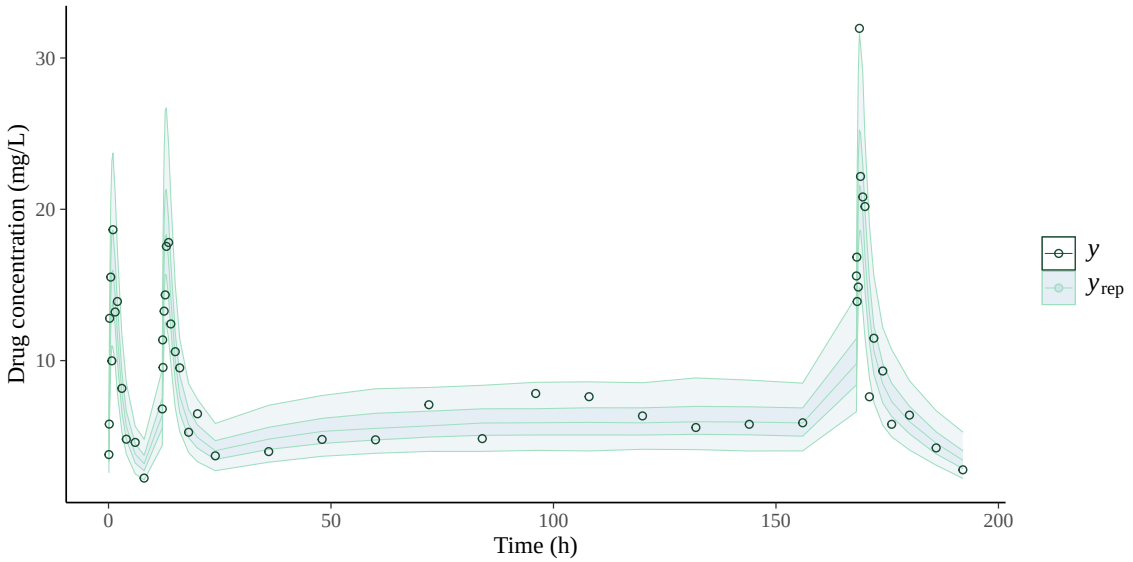

Figure 3: Posterior predictive checks for two compartment model. *The circles represent the observed data and the shaded areas the* 50^{th} *and* 90^{th} *credible intervals based on posterior draws.*

2.7 Comparing models: leave-one-out cross validation

Beyond model criticism, we may be interested in model comparison. Continuing our running example, we compare our two compartment model to a one compartment model, which is also supported by Torsten via the `pmx_solve_onecpt` routine. The corresponding posterior predictive checks are shown in Figure 4.

There are several ways of comparing models and which method is appropriate crucially depends on the insights we wish to gain. If our goal is to asses a model's ability to make good out-of-sample predictions, we may consider *Bayesian leave-one-out* (LOO) cross validation. The premise of cross-validation is to exclude a point, (y_i, x_i), from the *training set*, i.e. the set of data to which we fit the model. Here x_i denotes the covariate and in our example, the relevant row in the event schedule. We denote the reduced data set, y_{-i}. We then generate a prediction $(\tilde{y}_i, x_i)$ using the fitted model, and compare $\tilde{y}_i$ to y_i. A classic metric to make this comparison is the squared error, $(\tilde{y}_i - y_i)^2$.

Another approach is to use the *LOO estimate of out-of-sample predictive fit*:

$$\text{elp}_{\text{loo}} := \sum_i^n \log p(y_i \mid y_{-i}).$$

Here, no prediction is made. We however examine how consistent an "unobserved" data point is with our fitted model. Computing this estimator is expensive, since it requires fitting the model to n different training sets in order to evaluate each term in the sum.

?] propose an estimator of elp_{loo}, which uses Pareto smooth importance

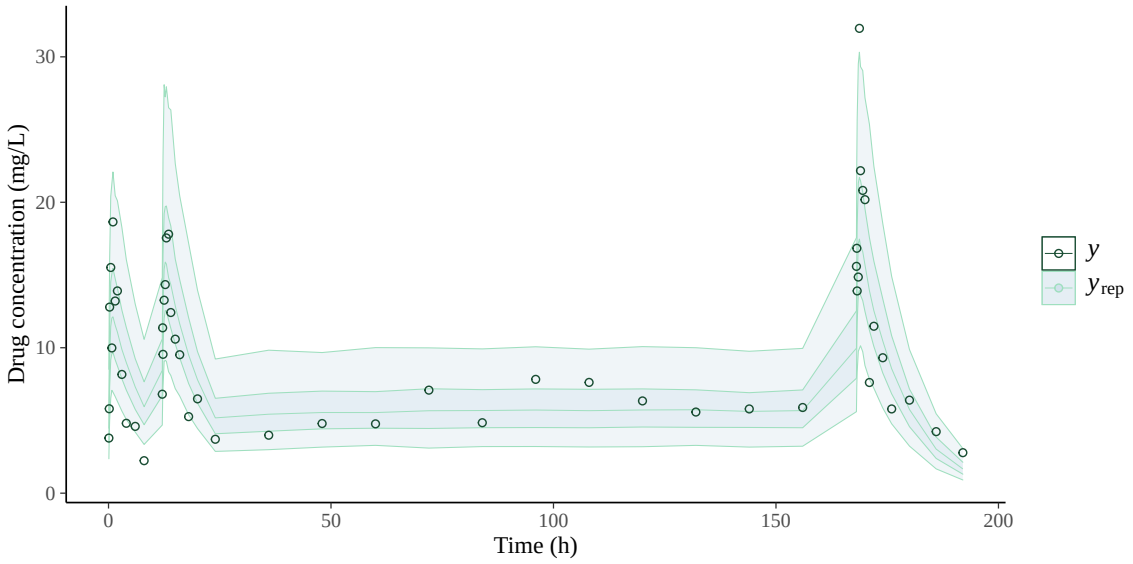

Figure 4: Posterior predictive checks for one compartment model. *The circles represent the observed data and the shaded areas the* 50^{th} *and* 90^{th} *credible intervals based on posterior draws. A graphical inspection suggests the credible interval is wider for the one compartment model than they are for the two compartment model.*

sampling and only requires a single model fit. The premise is to compute

$$\log p(y_i \mid y)$$

and correct this value, using importance sampling, to estimate $\log p(y_i \mid y_{-i})$. Naturally this estimator may be inaccurate. What makes this tool so useful is that we can use the Pareto shape parameter, $\hat{k}$, to asses how reliable the estimate is. In particular, if $\hat{k} > 0.7$, then the estimate shouldn't be trusted. The estimator is implemented in the R package Loo [**?**].

Conveniently, we can compute $\log p(y_i \mid y)$ in Stan's `generated quantities` block.

```
vector[nObs] log_lik;
for (i in 1:nObs)
  log_lik[i] =
    normal_lpdf(logCObs[i] | log(concentrationObs[i]),
                sigma);
```

These results can then be extracted and fed into Loo to compute elp_{loo}. The file `twoCpt.r` in the Supplementary Material shows exactly how to do this. Figure 5 plots the estimated elp_{loo}, along with a standard deviation, and shows the two compartment model has better out-of-sample predictive capabilities.

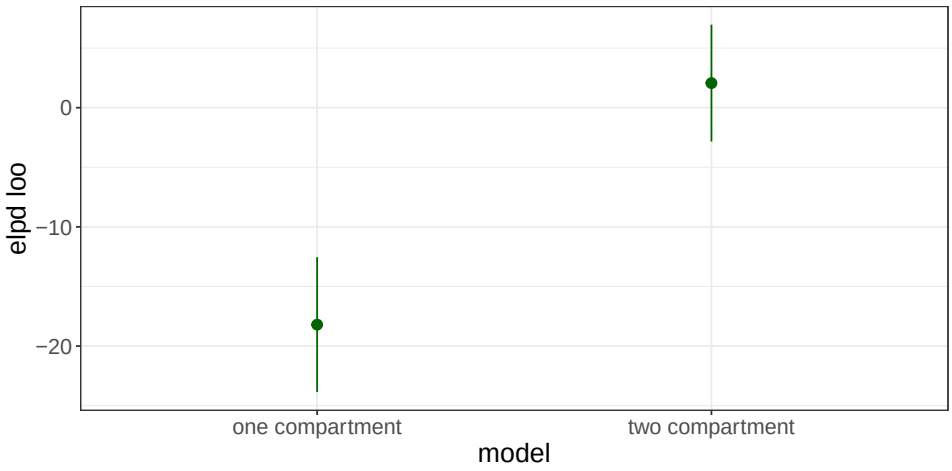

Figure 5: Leave-one-out estimate of out-of-sample predictive fit. *Plotted is the estimate,* elp_{loo}*, for the one and two compartment models. Clearly, the two compartment models has superior predictive capabilities.*

3 Two compartment population model

4 Non-linear pharmacokinetic / pharmacodynamic model

5 Conclusion

References

B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddel. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2017.