# Computational Statistics II

## Unit A.1: R programming and MCMC

**Tommaso Rigon**

**University of Milano-Bicocca**

Ph.D. in Economics and Statistics

# Unit A.1

## Main concepts

- Brief recap of inference via MCMC and the Metropolis–Hastings algorithm
- Weibull model with censored data
- Writing clean and efficient **R** code
- Reparametrizations and transformations.

- Associated **R** code: `https://tommasorigon.github.io/CompStat/exe/un_A1.html`

# Bayesian inference (recap)

- Let $\boldsymbol{X}$ be the data, following some distribution $\pi(\boldsymbol{X} \mid \boldsymbol{\theta})$, i.e. the **likelihood**, with $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^p$ being an unknown set of parameters.
- Let $\pi(\boldsymbol{\theta})$ be the **prior distribution** associated to $\boldsymbol{\theta}$.
- In Bayesian analysis, inference is based on the **posterior distribution** for $\boldsymbol{\theta}$, defined as

$$\pi(\boldsymbol{\theta} \mid \boldsymbol{X}) = \frac{\pi(\boldsymbol{\theta})\pi(\boldsymbol{X} \mid \boldsymbol{\theta})}{\int_\Theta \pi(\boldsymbol{\theta})\pi(\boldsymbol{X} \mid \boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}}.$$

- **Key issue**: the **normalizing constant**, i.e. the above integral, is often **intractable** $\implies$ no analytical solutions.
- Numerical approximations of $\int_\Theta \pi(\boldsymbol{\theta})\pi(\boldsymbol{X} \mid \boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}$ are highly unstable, especially in high dimensions $\implies$ the `integrate` **R** function will not work in most cases.

# Inference via MCMC (recap)

- **Key solution**. It is possible to **sample** from the posterior distribution even without knowing the normalizing constant.
- If we can get **random samples** $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(R)}$ from the posterior distribution, then we can approximate any functional of interest, i.e.

$$\mathbb{E}(f(\boldsymbol{\theta}) \mid \boldsymbol{X}) \approx \frac{1}{R} \sum_{r=1}^{R} f(\boldsymbol{\theta}^{(r)}).$$

- This approximation is often justified by the **ergodic theorem**.
- The samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(R)}$ are often dependent and may follow a Markov Chain $\Longrightarrow$ Markov Chain Monte Carlo (MCMC).

# Metropolis-Hastings algorithm (recap)

- A simple strategy for posterior sampling is the **Metropolis-Hastings** (MH) algorithm.
- Set the first value of the chain $\boldsymbol{\theta}_1$ to some (reasonable) value.

At the $r$th value of the chain

- Let $\boldsymbol{\theta} = \boldsymbol{\theta}^{(r)}$ be the current status of the chain.
- Sample $\boldsymbol{\theta}^*$ from a **proposal distribution** $q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})$.
- Compute the **acceptance probability**, defined as

$$\alpha = \min\left\{1, \frac{\pi(\boldsymbol{\theta}^* \mid \boldsymbol{X})}{\pi(\boldsymbol{\theta} \mid \boldsymbol{X})} \frac{q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^*)}{q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})}\right\} = \min\left\{1, \frac{\pi(\boldsymbol{\theta}^*)\pi(\boldsymbol{X} \mid \boldsymbol{\theta}^*)}{\pi(\boldsymbol{\theta})\pi(\boldsymbol{X} \mid \boldsymbol{\theta})} \frac{q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^*)}{q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta})}\right\}.$$

- With probability $\alpha$, **update the status** of the chain and set $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^*$.

# Implementation of MCMC

- The MH is perhaps the simplest MCMC algorithm and it has several limitations. We will discuss modifications / extensions of the MH in the next units.
- In units A.1-A.2 we will focus on **practical considerations** concerning the implementation with **R** and its interface with C++ using **Rcpp**.
- This is far from a comprehensive guide about **R** programming. We will consider a specific model and we will implement the relevant code in **R**.

## What about BUGS / JAGS / Stan?

- If the performance is not a concern, Stan-like software is an extremely useful tool for **practitioners** who wish to implement standard Bayesian models.
- Conversely, any non-standard or novel model, i.e. those usually developed by researchers in statistics, may be difficult or even impossible to implement.
- Besides, the "manual" implementation is very useful to **gain insights** about the model itself and it facilitates a lot the **debugging** process.

# Example: Weibull model for censored data

- We consider an example from survival analysis, i.e. the data are **survival times** which may be **censored**.
- In this example, we assume that the survival times are iid random variable following a Weibull distribution Weib$(\alpha, \beta)$.
- The observed survival time $t_i$ is either **complete** $(d_i = 1)$ or **right censored** $(d_i = 0)$, meaning that the survival time is higher than the observed $t_i$.
- The **hazard** and **survival** functions of a Weibull distribution are

$$h(t \mid \alpha, \beta) = \frac{\alpha}{\beta}\left(\frac{t}{\beta}\right)^{\alpha-1}, \quad S(t \mid \alpha, \beta) = \exp\left\{-\left(\frac{t}{\beta}\right)^{\alpha}\right\}.$$

- Recall that the **density** function is obtained as $f(t \mid \alpha, \beta) = h(t \mid \alpha, \beta)S(t \mid \alpha, \beta)$

# Likelihood function

- The **likelihood** for this parametric model, under suitable censorship assumptions, is **proportional** to the following quantity

$$\pi(\mathbf{t}, \mathbf{d} \mid \boldsymbol{\theta}) \propto \prod_{i=1}^{n} h(t_i \mid \alpha, \beta)^{d_i} S(t_i \mid \alpha, \beta) = \prod_{i:d_i=1} f(t_i \mid \alpha, \beta) \prod_{i:d_i=0} S(t_i \mid \alpha, \beta),$$

  with $(\alpha, \beta)$ being the parameter vector.

- **Remark** When performing (Bayesian) inference, note that the likelihood is always defined up to an **irrelevant normalizing constant** not depending on the parameters $\boldsymbol{\theta}$.

- These irrelevant constants **can and should be omitted** when performing computations, especially if they are expensive to evaluate.

# Bad implementation I (use the log-scale)

- In our experiments, we make use the `stanford2` dataset of the `survival` package.
- In first place, we need to implement the log-likelihood function, say `loglik`.
- The following implementation of the log-likelihood is correct but **numerically unstable**.

```
loglik_inaccurate <- function(t, d, alpha, beta) {
  hazard <- prod((alpha / beta * (t / beta)^(alpha - 1))^d)
  survival <- prod(exp(-(t / beta)^alpha))
  log(hazard * survival)
}

# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inaccurate(t, d, alpha = 0.5, beta = 1000)
# [1] -Inf
```

- The product of several terms close to 0 leads to numerical inaccuracies $\Longrightarrow$ **use the log-scale** instead.

# Bad implementation II (initialize the output)

- This second coding attempt relies on the log-scale and is indeed numerically much more stable than the previous version.
- However, this implementation is inefficient $\Longrightarrow$ **do not increase objects' dimension**.

```
loglik_inefficient2 <- function(t, d, alpha, beta) {
  n <- length(t) # Sample size
  log_hazards <- NULL
  log_survivals <- NULL

  for (i in 1:n) {
    log_hazards <- c(log_hazards, d[i] * ((alpha - 1) * log(t[i] / beta) + log(alpha / beta)))
    log_survivals <- c(log_survivals, -(t[i] / beta)^alpha)
  }
  sum(log_hazards) + sum(log_survivals)
}

# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inefficient2(t, d, alpha = 0.5, beta = 1000)
# [1] -873.3299
```

# Bad implementation III (avoid `for` loops)

- This third attempt avoids the previous pitfalls but it is still quite inefficient $\Longrightarrow$ **use vectorized code** whenever possible.

```
loglik_inefficient1 <- function(t, d, alpha, beta) {
  n <- length(t) # Sample size
  log_hazards <- numeric(n)
  log_survivals <- numeric(n)

  for (i in 1:n) {
    log_hazards[i] <- d[i] * ((alpha - 1) * log(t[i] / beta) + log(alpha / beta))
    log_survivals[i] <- -(t[i] / beta)^alpha
  }
  sum(log_hazards) + sum(log_survivals)
}

# Evaluate the log-likelihood at the point (0.5, 1000)
loglik_inefficient1(t, d, alpha = 0.5, beta = 1000)
# [1] -873.3299
```

# Good implementation

- The following version is both **numerically stable** and **efficient**.

```
loglik <- function(t, d, alpha, beta) {
  log_hazard <- sum(d * ((alpha - 1) * log(t / beta) + log(alpha / beta)))
  log_survival <- sum(-(t / beta)^alpha)
  log_hazard + log_survival
}

# Evaluate the log-likelihood at the point (0.5, 1000)
loglik(t, d, alpha = 0.5, beta = 1000)
# [1] -873.3299
```

- All these versions of `loglik` run in fractions of seconds. However, the `loglik` function must be executed i.e. $\sim 10^5$ times within a MH algorithm.

- Moreover, in more complex models several instances of these inefficiencies add up.

# Benchmarking the code

- To understand which function works better, you need to **test its performance**.
- There exist specialized packages to do so, i.e. **R** `rbenchmark` or `microbenchmark`.
- These packages execute the code several times and report the **average execution time**.
- The column "`elapsed`" refer to the overall time (in seconds) over 1000 replications.

```
library(rbenchmark) # Library for performing benchmarking

benchmark(
  loglik1 = loglik(t, d, alpha = 0.5, beta = 1000),
  loglik2 = loglik_inefficient1(t, d, alpha = 0.5, beta = 1000),
  loglik3 = loglik_inefficient2(t, d, alpha = 0.5, beta = 1000),
  columns = c("test", "replications", "elapsed", "relative"),
  replications = 1000
)

#       test replications elapsed relative
#1 loglik1         1000   0.014    1.000
#2 loglik2         1000   0.079    5.643
#3 loglik3         1000   0.412   29.429
```

# A matter of style

- **Formatting your code** properly is a healthy programming practice.
- You can refer to `https://style.tidyverse.org` for a comprehensive overview of good practices in **R**.
- Quoting the **tidyverse style guide**: "*Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread*".
- The `styler` **R** package automatically restyles your code for you and it is integrated within **RStudio** as an add-in.

```
# Good
x <- 5

# Bad
x = 5
```

# Reparametrizations I

- When performing (Bayesian) inference, the choice of the **parametrization** has strong impacts on computations.
- **General advice**: perform computations on the most convenient parametrization and then transform back the obtained samples.
- As a rule of thumb, you should use parametrizations with **unbounded domains**. This facilitates the choice of proposal distributions and could also **improve the mixing**.
- In our model, the two parameters $\alpha, \beta$ are strictly positive. Hence, a common strategy is to consider their logarithm, i.e. $\boldsymbol{\theta} = (\theta_1, \theta_2) = (\log \alpha, \log \beta)$.

## To log or not to log?

Roberts, G. O. and Rosenthal, J. S. (2009). *Examples of adaptive MCMC*. Journal of Computational and Graphical Statistics, **18**(2), 349–367.

# Reparametrizations & priors

- When reparametrizations are involved, there two possible modeling strategies.
- Choose the prior **before** the reparametrization. In our setting, we could let for example

$$\alpha \sim \text{Ga}(0.1, 0.1), \qquad \beta \sim \text{Ga}(0.1, 0.1).$$

  If you do so, remember to include the **jacobian** of the transformation when considering the transformed posterior!

- Choose the prior **after** the reparametrization. In our setting, we could let for example

$$\theta_1 = \log(\alpha) \sim \text{N}(0, 100), \qquad \theta_2 = \log(\beta) \sim \text{N}(0, 100).$$

  This strategy is simpler as it avoids the extra step of computing the jacobian.

```
logprior <- function(theta) {
  sum(dnorm(theta, 0, sqrt(100), log = TRUE))
}

logpost <- function(t, d, theta) {
  loglik(t, d, exp(theta[1]), exp(theta[2])) + logprior(theta)
}
```

# The MH implementation

- Since the space of $\boldsymbol{\theta}$ is unbounded, it is reasonable to select a **Gaussian random walk** as proposal distribution, namely

$$(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}) \sim \mathrm{N}_2(\boldsymbol{\theta}, 0.25^2 I_2).$$

  The choice of the variance will be discussed in unit B.1.

- Gaussian random walks are **symmetric** proposals distributions, implying that

$$q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^*) = q(\boldsymbol{\theta}^* \mid \boldsymbol{\theta}),$$

  which means that their ratio can be simplified $(= 1)$ when computing the acceptance probability $\alpha$.

- As before, make sure you compute $\alpha$ using the log-scale to avoid numerical instabilities.

- **Remark**. Unfortunately, there is no way to avoid `for` loops, which are highly inefficient $\Longrightarrow$ this justifies the usage of **Rcpp** and **RcppArmadillo**.

# Metropolis-Hastings code

```
RMH <- function(R, burn_in, t, d) {
  out <- matrix(0, R, 2) # Initialize an empty matrix to store the values
  theta <- c(0, 0) # Initial values
  logp <- logpost(t, d, theta) # Log-posterior
  for (r in 1:(burn_in + R)) {
    theta_new <- rnorm(2, mean = theta, sd = 0.25) # Propose a new value
    logp_new <- logpost(t, d, theta_new)
    alpha <- min(1, exp(logp_new - logp))
    if (runif(1) < alpha) {
      theta <- theta_new; logp <- logp_new # Accept the value
    }
    if (r > burn_in) {
      out[r - burn_in, ] <- theta     # Store the values after the burn-in period
    }
  }
  out
}
```

```
# Executing the code
library(tictoc) # Library for "timing" the functions
tic()
fit_MCMC <- RMH(R = 50000, burn_in = 5000, t, d)
toc()
# 0.92 sec elapsed
```

# Estimated survival function

- Posterior mean of the survival function with pointwise 95% credible intervals.