Item Response Model from scratch with Rcpp

vkyo23

February 14, 2022

Preface

In the repository, I have published the code to implement the Bayesian two parameter logistic item response model (2PL IRT) from scratch using `Rcpp` and `RcppArmadillo`.

In this RMarkdown file, I will describe the model and show an example of analysis using those codes[1].

IMPORTANT NOTE: As this is a transcription by a student who is still studying IRT for his own notes, there will be errors in various parts. Please let me know if there are any mistakes.

Likelihood and posteriors

Suppose there are $i = 1, \dots, N$ individuals and $j = 1, \dots, J$ items (or exams). The probability of an individual i's correct answer ($y_{ij} = 1$, otherwise 0) to an item j is given by

$$p_{ij} = \Pr(y_{ij} = 1 \mid \alpha_j, \beta_j, \theta_i) = \text{logit}^{-1}(\beta_j \theta_i - \alpha_j).$$

where $alpha_j$, β_j is the **difficulty** and **discrimination** of an item j, and θ_i is the ability of i. It is straightforward to show the likelihood f:

$$f(y \mid \alpha, \beta, \theta) = \prod_{i=1}^{N} \prod_{j=1}^{J} p_{ij}^{y_{ij}} (1 - p_{ij})^{1 - y_{ij}}. \tag{1}$$

For the above generative model, Bayesian estimation is used to estimate the latent variables, the parameters α_j, β_j and θ_i. From the Bayes rule, we can write the posterior distribution as

$$\pi(\Theta \mid y) \propto f(y \mid \Theta) \pi(\Theta), \tag{2}$$

where $\pi(\Theta)$ is **prior distribution**, $f(y \mid \Theta)$ is likelihood function and $\pi(\Theta \mid y)$ is **posterior distribution**.

Applying these to the above example, we will derive the posterior distribution of the item response theory model. First, we set prior distributions for α, β, θ

$$\alpha_j \sim N(a_0, A_0) \tag{3}$$
$$\beta_j \sim N(b_0, B_0) \tag{4}$$
$$\theta_i \sim N(0, 1). \tag{5}$$

[1] The description and implementation of model is relied on *Handbook of Item Response Theory Volume 2: Statistical Tools* by Wim J. van der Linden (2016). https://www.routledge.com/Handbook-of-Item-Response-Theory-Volume-2-Statistical-Tools/Linden/p/book/9780367221041

Then, we can write the conditional posterior distribution for α, β, θ from equation (1) ~ (5), given the rest parameters:

$$\begin{aligned}\pi(\alpha_j \mid y, \beta, \theta) &\propto f(y \mid \alpha, \beta, \theta)\pi(\alpha_j) \\ &= f(y \mid \alpha, \beta, \theta) \times N(\alpha_j \mid a_0, A_0) \\ &= \prod_{i=1}^{N} \left[p_{ij}^{y_{ij}}(1-p_{ij})^{1-y_{ij}}\right] N(\alpha_j \mid a_0, A_0) \quad \forall j = 1, \dots J.\end{aligned} \tag{6}$$

$$\begin{aligned}\pi(\beta_j \mid y, \alpha, \theta) &\propto f(y \mid \alpha, \beta, \theta)\pi(\beta_j) \\ &= f(y \mid \alpha, \beta, \theta) \times N(\beta_j \mid b_0, B_0) \\ &= \prod_{i=1}^{N} \left[p_{ij}^{y_{ij}}(1-p_{ij})^{1-y_{ij}}\right] N(\beta_j \mid b_0, B_0) \quad \forall j = 1, \dots J.\end{aligned} \tag{7}$$

$$\begin{aligned}\pi(\theta_i \mid y, \alpha, \beta) &\propto f(y \mid \alpha, \beta, \theta)\pi(\theta_i) \\ &= f(y \mid \alpha, \beta, \theta) \times N(\theta_i \mid 0, 1) \\ &= \prod_{j=1}^{J} \left[p_{ij}^{y_{ij}}(1-p_{ij})^{1-y_{ij}}\right] N(\theta_i \mid 0, 1) \quad \forall i = 1, \dots N.\end{aligned} \tag{8}$$

Normally, we would use a Gibbs sampler to sample parameters from the posterior distribution, but since the conditional posteriors in (6) ~ (8) above are not in the form of the standard normal or gamma distributions, we cannot simply sample parameters using a Gibbs sampler. Therefore, we adopt the Metropolis-Hastings Algorithm, which allows us to perform MCMC even in such a case.

Metropolis-Hastings Algorithm

Here, for simplicity of notation, I define an arbitrary parameter as δ_k. First, we sample the **candidate** δ_k^* from a random walk distribution as follows

$$\delta_k^* \sim N(\delta_k^{(t-1)}, \tau_\delta),$$

where $\delta_k^{(t-1)}$ is a sample from previous iteration and τ is a so-called tuning parameter. Next, we must calculate **the acceptance probability** which determines whether we accept the candidate δ_k^* or previous $\delta_k^{(t-1)}$ as a current sample. Following the discussion of Junker, Patz and VanHoudnos (2016, p.277)[2], the acceptance probability is given by:

$$ap = \min\left\{\frac{\pi(\delta_k^* \mid y, \eta) \cdot g(\delta_k^{(t-1)} \mid \delta_k^*)}{\pi(\delta_k^{(t-1)} \mid y, \eta) \cdot g(\delta_k^* \mid \delta_k^{(t-1)})}, 1\right\}$$

where $\pi(\cdot \mid y, \eta)$ is the posterior density, $g(\cdot \mid \cdot)$ is proposal density and η indicates rest parameters other than δ_k. If $u < ap$, we accept δ_k^* as a current sample, otherwise $\delta_k^{(t-1)}$ where $u \sim U(0, 1)$.

To calculate the acceptance probability, we first calculate the posterior density of the candidate sample, $\pi(\delta_k^* \mid y, \eta)$, and the previous sample, $\pi(\delta_k^{(t-1)} \mid y, \eta)$, respectively. For example, the case of α_j:

[2] From the same book as in footnote 1.

$$
\begin{aligned}
\text{Candidate} \quad \ldots \quad \pi(\alpha_j^* \mid y, \beta, \theta) &\propto f(y \mid \alpha^*, \beta, \theta)\pi(\alpha_j^*) \\
&= f(y \mid \alpha^*, \beta, \theta) \times N(\alpha_j^* \mid a_0, A_0) \\
&= \prod_{i=1}^{N} \left[p_{ij}(\alpha_j^*)^{y_{ij}} \{1 - p_{ij}(\alpha_j^*)\}^{1-y_{ij}}\right] N(\alpha_j^* \mid a_0, A_0) \quad \forall j = 1, \ldots J.
\end{aligned}
$$

$$
\begin{aligned}
\text{Previous} \quad \ldots \quad \pi(\alpha_j^{(t-1)} \mid y, \beta, \theta) &\propto f(y \mid \alpha^{(t-1)}, \beta, \theta)\pi(\alpha_j^{(t-1)}) \\
&= f(y \mid \alpha^{(t-1)}, \beta, \theta) \times N(\alpha_j^{(t-1)} \mid a_0, A_0) \\
&= \prod_{i=1}^{N} \left[p_{ij}(\alpha_j^{(t-1)})^{y_{ij}} \{1 - p_{ij}(\alpha_j^{(t-1)})\}^{1-y_{ij}}\right] N(\alpha_j^{(t-1)} \mid a_0, A_0) \quad \forall j = 1, \ldots J.
\end{aligned}
$$

Next, we calculate the proposal density $g(\alpha_j^* \mid \alpha_j^{(t-1)})$ and $g(\alpha_j^{(t-1)} \mid \alpha_j^*)$

$$
\begin{aligned}
\text{Candidate} \quad \cdots \quad g(\alpha_j^* \mid \alpha_j^{(t-1)}) &= N(\alpha_j^* \mid \alpha_j^{(t-1)}, \tau_\alpha) \quad \forall \quad j = 1, \ldots, J \\
\text{Previous} \quad \cdots \quad g(\alpha_j^{(t-1)} \mid \alpha_j^*) &= N(\alpha_j^{(t-1)} \mid \alpha_j^*, \tau_\alpha) \quad \forall \quad j = 1, \ldots, J.
\end{aligned}
$$

The same steps are applied to the case of other parameters β_j and θ_i.

Here, we convert the acceptance probability into natural logarithm for the convenience of computation, that is:

$$
\log(ap) = \min\left\{\log[\pi(\delta_k^* \mid y, \eta)] + \log[g(\delta_k^{(t-1)} \mid \delta_k^*)] - \log[\pi(\delta_k^{(t-1)} \mid y, \eta)] - \log[g(\delta_k^* \mid \delta_k^{(t-1)}], 0\right\},
$$

and if $\log(u) < \log(ap)$, we accept δ_k^* otherwise $\delta_k^{(t-1)}$.

Coding the sampler

In this section, I write down the code for parameter sampling.

Log-liklihood calculator `loglik.cpp`

First, I introduce helper function `loglik` which is enable us to calculate log-likelihood. The log-likelihood function is written as

$$
l(y \mid \alpha, \beta, \theta) = \sum_{i=1}^{N} \sum_{j=1}^{J} [y_{ij} \log(p_{ij}) + (1 - y_{ij}) \log(1 - p_{ij})].
$$

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;
using namespace Rcpp;

// Log-likelihood function
```

```
arma::mat loglik(arma::mat Y, arma::vec alpha,
                 arma::vec beta, arma::vec theta) {
  //calculate beta_j * theta_i
  arma::mat temp = beta * theta.t();

  //beta_j * theta_i - alpha_j
  arma::mat temp2 = temp.each_col() - alpha;

  //exp(beta_j * theta_i - alpha_j)
  arma::mat exp_ = arma::exp(temp2.t());

  //inverse logit
  arma::mat p = exp_ / (1 + exp_);

  //calculate log-lik
  arma::mat log_lik = Y % arma::log(p) + (1 - Y) % arma::log(1 - p);

  return(log_lik);
}
```

Sampling α_j `alpha_sample.cpp`

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include "loglik.h"
using namespace arma;
using namespace Rcpp;

// SAMPLING ALPHA
NumericVector alpha_sample(arma::mat Y, arma::vec alpha_old, arma::vec beta_old,
                           arma::vec theta_old, double a0, double A0, double MH_alpha) {
  // NOTE:
  // _star -> candidate
  // _old -> previous

  int J = Y.n_cols; //# of alpha
  arma::vec alpha_star(J); //candidate sample for alpha
  arma::vec log_prop_star(J); //log proposal density for alpha_star
  arma::vec log_prop_old(J); //log proposal density for alpha_old

  //Sample theta_star and log proposal density.
  for (int j = 0; j < J; j++) {
    alpha_star[j] = R::rnorm(alpha_old[j], MH_alpha);
    log_prop_star[j] = R::dnorm(alpha_star[j], alpha_old[j], MH_alpha, true);
    log_prop_old[j] = R::dnorm(alpha_old[j], alpha_star[j], MH_alpha, true);
  }

  //log-likelihood
  arma::rowvec loglik_star = colSums(as<NumericMatrix>(wrap(loglik(Y, alpha_star, beta_old, theta_old)))
                                     true);
  arma::rowvec loglik_old = colSums(as<NumericMatrix>(wrap(loglik(Y, alpha_old, beta_old, theta_old))),
```

```
                                    true);

  //log prior density
  arma::rowvec log_dnorm_star =  dnorm(as<NumericVector>(wrap(alpha_star)), a0, A0, true);
  arma::rowvec log_dnorm_old = dnorm(as<NumericVector>(wrap(alpha_old)), a0, A0, true);

  //log posterior density
  arma::rowvec log_pd_star = loglik_star + log_dnorm_star;
  arma::rowvec log_pd_old = loglik_old + log_dnorm_old;

  //log acceptance probability
  arma::vec log_densfrac = log_pd_star.t() + log_prop_old - log_pd_old.t() - log_prop_star;
  NumericVector log_ap = pmin(as<NumericVector>(wrap(log_densfrac)), 0);
  NumericVector log_u = log(runif(J, 0, 1));

  //save samples
  NumericVector sample = ifelse(log_u < log_ap, as<NumericVector>(wrap(alpha_star)),
                                as<NumericVector>(wrap(alpha_old)));

  return(sample);
}
```

Sampling β_j `beta_sample.cpp`

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include "loglik.h"
using namespace arma;
using namespace Rcpp;

// SAMPLING BETA
NumericVector beta_sample(arma::mat Y, arma::vec alpha_old, arma::vec beta_old,
                          arma::vec theta_old, double b0, double B0, double MH_beta) {

  // NOTE:
  // _star -> candidate
  // _old -> previous

  int J = Y.n_cols; //# of beta
  arma::vec beta_star(J); //candidate sample for beta
  arma::vec log_prop_star(J); //log proposal density for beta_star
  arma::vec log_prop_old(J); //log proposal density for beta_old

  //Sample theta_star and log proposal density.
  for (int j = 0; j < J; j++) {
    beta_star[j] = R::rnorm(beta_old[j], MH_beta);
    log_prop_star[j] = R::dnorm(beta_star[j], beta_old[j], MH_beta, true);
    log_prop_old[j] = R::dnorm(beta_old[j], beta_star[j], MH_beta, true);
  }

  //log-likelihood
```

```
  arma::rowvec loglik_star = colSums(as<NumericMatrix>(wrap(loglik(Y, alpha_old, beta_star, theta_old))
                                     true);
  arma::rowvec loglik_old = colSums(as<NumericMatrix>(wrap(loglik(Y, alpha_old, beta_old, theta_old))),
                                    true);

  //log prior density
  arma::rowvec log_dnorm_star =  dnorm(as<NumericVector>(wrap(beta_star)), b0, B0, true);
  arma::rowvec log_dnorm_old = dnorm(as<NumericVector>(wrap(beta_old)), b0, B0, true);

  //log posterior density
  arma::rowvec log_pd_star = loglik_star + log_dnorm_star;
  arma::rowvec log_pd_old = loglik_old + log_dnorm_old;

  //log acceptance probability
  arma::vec log_densfrac = log_pd_star.t() + log_prop_old - log_pd_old.t() - log_prop_star;
  NumericVector log_ap = pmin(as<NumericVector>(wrap(log_densfrac)), 0);
  NumericVector log_u = log(runif(J, 0, 1));

  //save samples
  NumericVector sample = ifelse(log_u < log_ap, as<NumericVector>(wrap(beta_star)),
                                as<NumericVector>(wrap(beta_old)));

  return(sample);
}
```

Sampling θ_i `theta_sample.cpp`

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include "loglik.h"
using namespace arma;
using namespace Rcpp;

// SAMPLING THETA
NumericVector theta_sample(arma::mat Y, arma::vec alpha_old, arma::vec beta_old,
                           arma::vec theta_old, double MH_theta) {

  // NOTE:
  // _star -> candidate
  // _old -> previous

  int I = Y.n_rows; //# of theta
  arma::vec theta_star(I); //candidate sample for theta
  arma::vec log_prop_star(I); //log proposal density for theta_star
  arma::vec log_prop_old(I); //log proposal density for theta_old

  //Sample theta_star and log proposal density.
  for (int i = 0; i < I; i++) {
    theta_star[i] = R::rnorm(theta_old[i], MH_theta);
    log_prop_star[i] = R::dnorm(theta_star[i], theta_old[i], MH_theta, true);
    log_prop_old[i] = R::dnorm(theta_old[i], theta_star[i], MH_theta, true);
```

```
  }

  //log-likelihood
  arma::vec loglik_star = rowSums(as<NumericMatrix>(wrap(loglik(Y, alpha_old, beta_old, theta_star))),
                                  true);
  arma::vec loglik_old = rowSums(as<NumericMatrix>(wrap(loglik(Y, alpha_old, beta_old, theta_old))),
                                 true);

  //log prior density
  arma::vec log_dnorm_star =  dnorm(as<NumericVector>(wrap(theta_star)), 0, 1, true);
  arma::vec log_dnorm_old = dnorm(as<NumericVector>(wrap(theta_old)), 0, 1, true);

  //log posterior density
  arma::vec log_pd_star = loglik_star + log_dnorm_star;
  arma::vec log_pd_old = loglik_old + log_dnorm_old;

  //log acceptance probability
  arma::vec log_densfrac = log_pd_star + log_prop_old - log_pd_old - log_prop_star;
  NumericVector log_ap = pmin(as<NumericVector>(wrap(log_densfrac)), 0);
  NumericVector log_u = log(runif(I, 0, 1));

  //save samples
  NumericVector sample = ifelse(log_u < log_ap, as<NumericVector>(wrap(theta_star)),
                                as<NumericVector>(wrap(theta_old)));

  return(sample);
}
```

Sampler `sampler_irt.cpp`

In MCMCpack, reparameterization of estimated parameters is done. Specifically, θ_i is standardized with mean 0 and sd 1, and α_j, β_j are alsop adjusted accordingly. Specifically, this is.

$$
\begin{aligned}
\theta_i^{adj} &= \frac{\theta_i - \overline{\theta}}{s_\theta} \\
\alpha_j^{adj} &= \beta_j \overline{\theta} - \alpha_j \\
\beta_j^{adj} &= \beta_j s_\theta,
\end{aligned}
$$

where s_θ is standard deviation of θ and $\overline{\theta}$ is the mean. In my sampler, I have also incorporated these processes properly.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include "alpha_sample.h"
#include "beta_sample.h"
#include "theta_sample.h"
using namespace Rcpp;
using namespace arma;

// SAMPLER
```

```
// [[Rcpp::export]]
List sampler_irt(arma::mat datamatrix, arma::vec alpha, arma::vec beta,
                 arma::vec theta, double a0, double A0,
                 double b0, double B0,
                 double MH_alpha, double MH_beta, double MH_theta,
                 int iter, int warmup, int thin, int refresh) {

  int total_iter = iter + warmup; // total iteration
  int sample_iter = iter / thin; // # of samples to save

  arma::mat Y = datamatrix; // rename datamatrix to Y
  int I = Y.n_rows; // # of individuals
  int J = Y.n_cols; // # of items

  // rename
  arma::vec alpha_old = alpha;
  arma::vec beta_old = beta;
  arma::vec theta_old = theta;

  // create storages for parameters
  NumericMatrix theta_store(I, sample_iter);
  NumericMatrix alpha_store(J, sample_iter);
  NumericMatrix beta_store(J, sample_iter);


  // WARMUP
  Rcout << "Warmup:   " << 1 << " / " << total_iter << " [ " << 0 << "% ]\n";
  for (int g = 0; g < warmup; g++) {
    if ((g + 1) % refresh == 0) {
      double gg = g + 1;
      double ti2 = total_iter;
      double per = std::round((gg / ti2) * 100);
      Rcout << "Warmup:   " << (g + 1) << " / " << total_iter << " [ " << per << "% ]\n";
    }
    theta = theta_sample(Y, alpha_old, beta_old, theta_old, MH_theta);
    theta_old = theta;
    alpha = alpha_sample(Y, alpha_old, beta_old, theta_old, a0, A0, MH_alpha);
    alpha_old = alpha;
    beta = beta_sample(Y, alpha_old, beta_old, theta_old, b0, B0, MH_beta);
    beta_old = beta;
  }

  // SAMPLING
  double gg = warmup + 1;
  double ti2 = total_iter;
  double per = std::round((gg / ti2) * 100);
  Rcout << "Sampling: " << gg << " / " << total_iter << " [ " << per << "% ]\n";
  for (int g = warmup; g < total_iter; g++) {
    if ((g + 1) % refresh == 0) {
      double gg = g + 1;
      double ti2 = total_iter;
      double per = std::round((gg / ti2) * 100);
      Rcout << "Sampling: " << (g + 1) << " / " << total_iter << " [ " << per << " % ]\n";
```

```
    }
    theta = theta_sample(Y, alpha_old, beta_old, theta_old, MH_theta);
    theta_old = theta;
    alpha = alpha_sample(Y, alpha_old, beta_old, theta_old, a0, A0, MH_alpha);
    alpha_old = alpha;
    beta = beta_sample(Y, alpha_old, beta_old, theta_old, b0, B0, MH_beta);
    beta_old = beta;

    if (g % thin == 0) {
      double th = thin;
      double wu = warmup;
      double gg = g;
      double ggg = (g - warmup) / thin;

      // Reparameterization (fix theta with mean 0 and sd 1)

      NumericVector theta_nv = as<NumericVector>(wrap(theta_old));
      NumericVector alpha_nv = as<NumericVector>(wrap(alpha_old));
      NumericVector beta_nv = as<NumericVector>(wrap(beta_old));

      NumericVector theta_std = (theta_nv - mean(theta_nv)) / sd(theta_nv);
      NumericVector alpha_std = beta_nv * mean(theta_nv) - alpha_nv;
      NumericVector beta_std = beta_nv * sd(theta_nv);

      theta_store(_, ggg) = theta_std;
      alpha_store(_, ggg) = alpha_std;
      beta_store(_, ggg) = beta_std;
    }
  }

  List L = List::create(Named("alpha") = alpha_store,
                        Named("beta") = beta_store,
                        Named("theta") = theta_store);
  return(L);
}
```

R wrapper function

I also write a wrapper function `irt_cpp` for the sampler.

```
# This code is from "irt_cpp.R"
irt_cpp <- function(datamatrix, iter = 2000, warmup = 1000, thin = 1, refresh = 100,
                    seed, init, tuning_par, prior) {

  # datamatrix -> individual * item matrix (matrix)
  # iter -> # of iterations (int)
  # warmup -> # of burn-in (int)
  # thin -> Save sample every [thin] iteration (int)
  # refresh -> Output the status of sampling every [refresh] iteration (int)
  # seed -> seed value (double)
  # init -> initial values (list, please name correctly as below!)
  #         -> alpha: init for alpha
```

```r
#         -> beta: init for beta
#         -> theta: init for theta
# tuning_par -> tuning parameter (list, please name correctly as below!)
#               -> alpha: tau for alpha
#               -> beta: tau for beta
#               -> theta: tau for theta
#
# prior -> priors (list, please name correctly as below!)
#         -> a0: prior means for alpha
#         -> A0: prior sd for alpha
#         -> b0: prior means for beta
#         -> B0: prior sd for beta

cat("\n================================================================\n")
cat("Run Metropolis-Hasting Sampler for 2PL item response model...\n\n")
cat("  Observations:", nrow(datamatrix) * ncol(datamatrix),"\n")
cat("    Number of individuals:", nrow(datamatrix), "\n")
cat("    Number of items:", ncol(datamatrix), "\n")
cat("    Total correct response:", sum(as.numeric(datamatrix), na.rm = TRUE), "/",
    nrow(datamatrix) * ncol(datamatrix),
    "[", round(sum(as.numeric(datamatrix), na.rm = TRUE) / (nrow(datamatrix) *
                                                              ncol(datamatrix)), 2) * 100, "%]","\n\n
cat("  Priors: \n")
cat("    alpha ~", paste0("N(", prior$a0, ", ", prior$A0, "),"),
    "beta ~", paste0("N(", prior$b0, ", ", prior$B0, "),"),
    "theta ~ N(0, 1).\n")
cat("==================================================================\n\n")

# Preparation
## Measure starting time
stime <- proc.time()[3]
## Set seed
set.seed(seed)

# Run sampler
mcmc <- sampler_irt(datamatrix = Y,
                    alpha = init$alpha,
                    beta = init$beta,
                    theta = init$theta,
                    a0 = prior$a0,
                    A0 = prior$A0,
                    b0 = prior$b0,
                    B0 = prior$B0,
                    MH_alpha = tuning_par$alpha,
                    MH_beta = tuning_par$beta,
                    MH_theta = tuning_par$theta,
                    iter = iter,
                    warmup = warmup,
                    thin = thin,
                    refresh = refresh)

# Generate variable labels
label_iter <- paste0("iter_", 1:(iter/thin))
```

```r
  alpha_lab <- paste0("alpha_", colnames(datamatrix))
  beta_lab <- paste0("beta_", colnames(datamatrix))
  theta_lab <- paste0("theta_", rownames(datamatrix))
  colnames(mcmc$alpha) <- colnames(mcmc$beta) <- colnames(mcmc$theta) <- label_iter
  rownames(mcmc$alpha) <- alpha_lab
  rownames(mcmc$beta) <- beta_lab
  rownames(mcmc$theta) <- theta_lab

  # Redefine quantile function
  lwr <- function(x) quantile(x, probs = 0.025, na.rm = TRUE)
  upr <- function(x) quantile(x, probs = 0.975, na.rm = TRUE)
  mean_ <- function(x) mean(x, na.rm = TRUE)
  median_ <- function(x) median(x, na.rm = TRUE)

  # Calculate statistics
  alpha_post <- data.frame(parameter = alpha_lab,
                           mean = apply(mcmc$alpha, 1, mean_),
                           median = apply(mcmc$alpha, 1, median_),
                           lwr = apply(mcmc$alpha, 1, lwr),
                           upr = apply(mcmc$alpha, 1, upr))
  beta_post <- data.frame(parameter = beta_lab,
                          mean = apply(mcmc$beta, 1, mean_),
                          median = apply(mcmc$beta, 1, median_),
                          lwr = apply(mcmc$beta, 1, lwr),
                          upr = apply(mcmc$beta, 1, upr))
  theta_post <- data.frame(parameter = theta_lab,
                           mean = apply(mcmc$theta, 1, mean_),
                           median = apply(mcmc$theta, 1, median_),
                           lwr = apply(mcmc$theta, 1, lwr),
                           upr = apply(mcmc$theta, 1, upr))

  # Aggregate
  result <- list(summary = list(alpha = alpha_post,
                                beta = beta_post,
                                theta = theta_post),
                 sample = list(alpha = mcmc$alpha,
                               beta = mcmc$beta,
                               theta = mcmc$theta))
  etime <- proc.time()[3]
  cat(crayon::yellow("Done: Total time", round(etime - stime, 1), "sec\n"))
  return(result)
}
```

Example: 106th US Senate roll-call vote analysis

As an example, I will apply item response theory using data of the 106th US Senate roll-call vote. The data is from the {MCMCpack} package[3]. In the field of political science, IRT is frequently used to measure the policy positions (a.k.a. ideal points) of political actors using roll call voting and judgment data. For a discussion of the relevance of spatial voting models to IRT, see Clinton, Jackman & Rivers (2004, APSR)[4].

[3] See more detail: https://cran.r-project.org/web/packages/MCMCpack/MCMCpack.pdf

[4] Clinton, J., Jackman, S., & Rivers, D. (2004). The statistical analysis of roll call data. American Political Science Review, 98(2), 355-370.

First, load {Rcpp} package, compile `sampler_irt.cpp` and data.

```
library(Rcpp)
sourceCpp("cpp/sampler_irt.cpp")
data(Senate, package = "MCMCpack")
```

To run the sampler, we must convert the data into **roll-call matrix (individual * item matrix)**.

```
# Check data (data frame)
Senate[1:10, 1:10]
```

```
              id statecode   state party     member rc1 rc2 rc3 rc4 rc5
SESSIONS   49700        41 ALABAMA     1   SESSIONS   1   0   0   0   1
SHELBY     94659        41 ALABAMA     1     SHELBY   1   0   0   0   1
MURKOWSKI  14907        81  ALASKA     1  MURKOWSKI   1   0   0   0   1
STEVENS    12109        81  ALASKA     1    STEVENS   1   0   0   0   1
KYL        15429        61 ARIZONA     1        KYL   1   0   0   0   1
MCCAIN     15039        61 ARIZONA     1     MCCAIN   1   0   0   0   1
HUTCHINSON 29306        42 ARKANSA     1 HUTCHINSON   1   0   0   0   1
 [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

```
# → unnecessary variables are recorded, so drop

# Drop some variables and convert into matrix
Y <- as.matrix(Senate[, 6:ncol(Senate)])
```

Also, we set **initial values** for sampling, **tuning parameters** and **priors**. In this analysis, I set the priors as follows:

$$\alpha_j \sim N(0, 10), \quad \beta_j \sim N(1, 0.2).$$

Then, $a_0 = 0, A_0 = 10, b_0 = 1, B_0 = 0.2$. And I supply very flat initial values – all inits are set 0.1.

```
# Note: We must name the list of element correctly as below to insert the list to wrapper function
# Initial values
init <- list(alpha = rep(0.1, ncol(Y)),
             beta = rep(0.1, ncol(Y)),
             theta = rep(0.1, nrow(Y)))

# Tuning parameters
tuning_par <- list(alpha = 0.4,
                   beta = 0.4,
                   theta = 0.4)

# Priors
prior <- list(a0 = 0, # prior mean for alpha
              A0 = 10, # prior sd for alpha
              b0 = 1, # prior mean for beta
              B0 = 0.2) # prior sd for beta
```

Yes! We are ready to run MCMC!

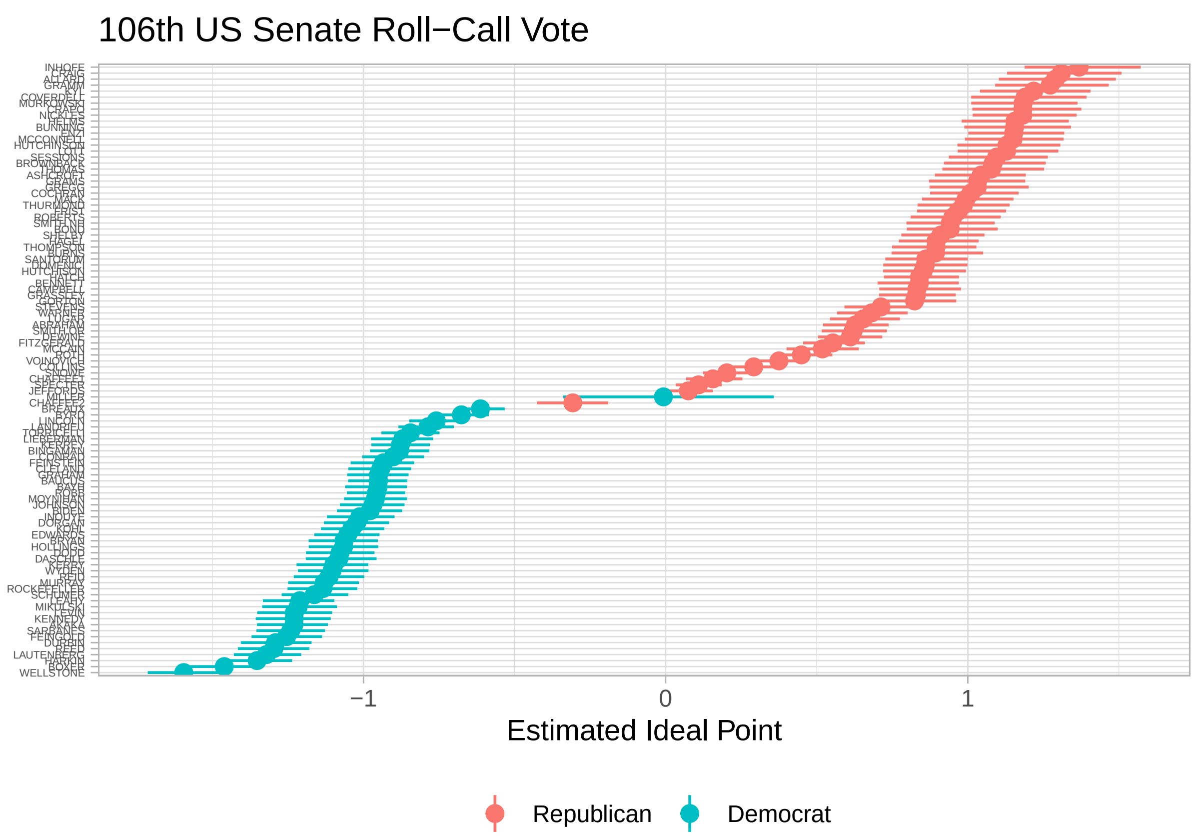
```r
# Run
fit <- irt_cpp(datamatrix = Y, # roll-call matrix
               iter = 5000, # number of sampling
               warmup = 3000, # burn-in
               thin = 5, # save samples every [thin] steps
               refresh = 1000, # output the status of sampling every [refresh] steps
               seed = 1, # seed value
               init = init, # initial values
               tuning_par = tuning_par, # tuning parameters
               prior = prior) # priors
```

```
================================================================
Run Metropolis-Hasting Sampler for 2PL item response model...

  Observations: 68544
    Number of individuals: 102
    Number of items: 672
    Total correct response: 42458 / 68544 [ 62 %]

  Priors:
    alpha ~ N(0, 10), beta ~ N(1, 0.2), theta ~ N(0, 1).
================================================================

Warmup:   1 / 8000 [ 0% ]
Warmup:   1000 / 8000 [ 13% ]
Warmup:   2000 / 8000 [ 25% ]
Warmup:   3000 / 8000 [ 38% ]
Sampling: 3001 / 8000 [ 38% ]
Sampling: 4000 / 8000 [ 50 % ]
Sampling: 5000 / 8000 [ 63 % ]
Sampling: 6000 / 8000 [ 75 % ]
Sampling: 7000 / 8000 [ 88 % ]
Sampling: 8000 / 8000 [ 100 % ]
Done: Total time 191.7 sec
```

Foo! Sampler finished. Next, we extract the result and plot the senators' ideal point.

```r
# For data handling
library(tidyverse)

# Extract the result
theta <- fit$summary$theta %>%
  mutate(name = rownames(Y),
         party = Senate$party)

# Plot
theta %>%
  ggplot(aes(y = reorder(name, mean), x = mean, color = factor(party))) +
  geom_pointrange(aes(xmin = lwr, xmax = upr)) +
  theme_light() +
  xlab("Estimated Ideal Point") +
  ylab("") +
  ggtitle("106th US Senate Roll-Call Vote") +
```

```
  scale_color_discrete(limits = c("1", "0"),
                       label = c("Republican", "Democrat")) +
  theme(legend.position = "bottom",
        legend.direction = "horizontal",
        legend.title = element_blank(),
        axis.text.y = element_text(size = 4))
```



We have clear estimate of the policy positions of each senator, well divided by party. Just to be sure, let's use `MCMCpack::MCMCirt1d` to see if our estimates is similar to it.

```
stime <- proc.time()[3]
fit_mcmc <- MCMCpack::MCMCirt1d(Y,
                                mcmc = 5000,
                                burnin = 3000,
                                thin = 5,
                                verbose = 1000)
```

```
MCMCirt1d iteration 1 of 8000
```

```
MCMCirt1d iteration 1001 of 8000
```

```
MCMCirt1d iteration 2001 of 8000
```

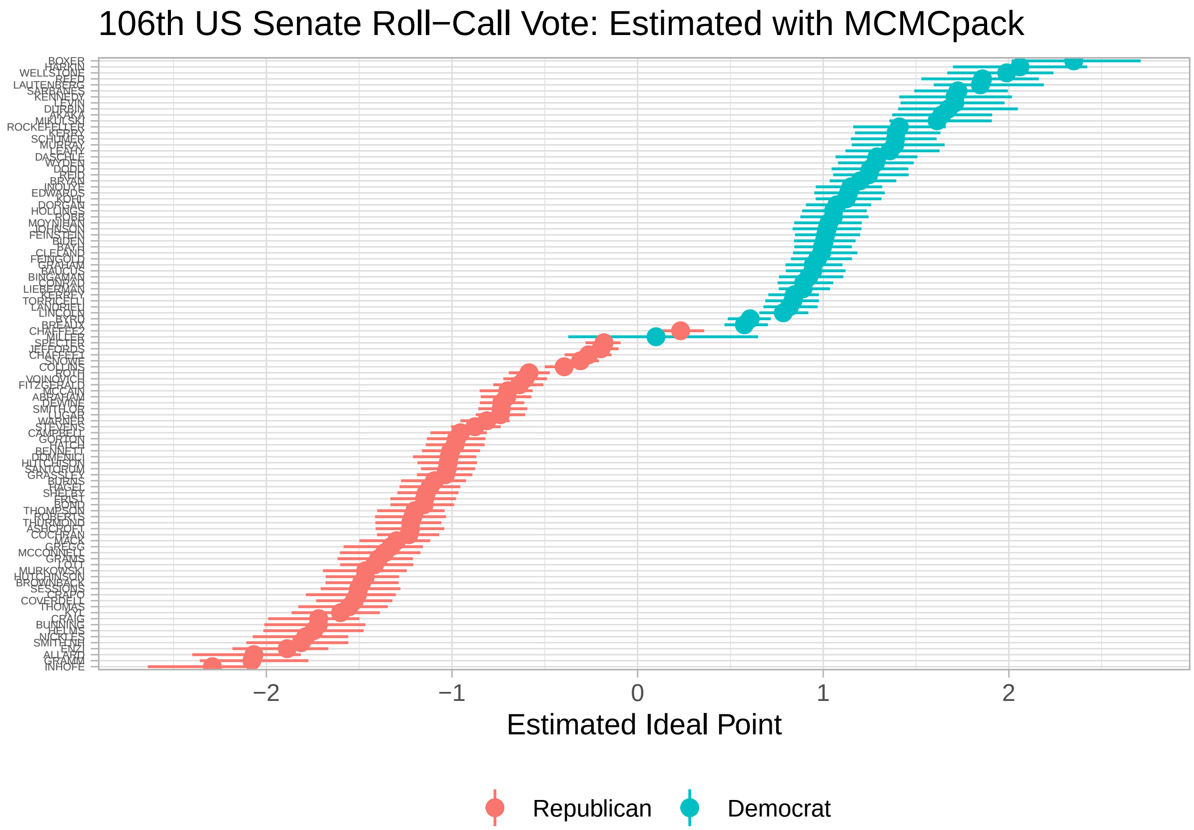
```
MCMCirt1d iteration 3001 of 8000


MCMCirt1d iteration 4001 of 8000


MCMCirt1d iteration 5001 of 8000


MCMCirt1d iteration 6001 of 8000


MCMCirt1d iteration 7001 of 8000
```

```r
proc.time()[3] - stime
```

```
elapsed
 55.007
```

Very unfortunately, it is much faster to do IRT with MCMCpack. Also, it seems that MCMCpack uses probit execution instead of logit. I don't know if that affects the execution time, but it does seem very fast. Let's check the result.

```r
theta_mcmc <- summary(fit_mcmc)$quantiles
theta_mcmc %>%
  as_tibble() %>%
  mutate(name = rownames(Y),
         party = Senate$party,
         mean = `50%`,
         lwr = `2.5%`,
         upr = `97.5%`) %>%
  ggplot(aes(y = reorder(name, mean), x = mean, color = factor(party))) +
  geom_pointrange(aes(xmin = lwr, xmax = upr)) +
  theme_light() +
  xlab("Estimated Ideal Point") +
  ylab("") +
  ggtitle("106th US Senate Roll-Call Vote: Estimated with MCMCpack") +
  scale_color_discrete(limits = c("1", "0"),
                       label = c("Republican", "Democrat")) +
  theme(legend.position = "bottom",
        legend.direction = "horizontal",
        legend.title = element_blank(),
        axis.text.y = element_text(size = 4))
```



The sign of θ_i is reversed, but both estimates are similar!

So, my sampler works well!!!!!!!!!