State-space model representation of univariate Gaussian process regression

Xinrui Shi

2023-06-02

Functions for inference of GP using Kalman Filter

- **Input:** (m_0, σ_0^2).
- **Output:** $\left\{(m_{t|t}, \sigma_{t|t}^2), (m_{t+1|t}, \sigma_{t+1|t}^2)\right\}_{t=1}^{T}$.

1. initialization: use the provided (m_0, σ_0^2) to estimate $(m_{1|1}, \sigma_{1|1}^2)$, such that:

$$m_{1|1} = \frac{\sigma_0^2}{\sigma_w^2 + \sigma_0^2}(y_1 - m_0), \qquad \sigma_{1|1}^2 = \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_w^2}.$$

2. for $t = 1, \ldots, T-1$:

   - prediction step: estimate $(m_{t+1|t}, \sigma_{t+1|t}^2)$, such that:

$$m_{t+1|t} = \exp\left(-\frac{1}{\gamma}\right) m_{t|t}, \qquad \sigma_{t+1|t}^2 = \left(1 - \exp\left(-\frac{2}{\gamma}\right)\right) + \exp\left(-\frac{2}{\gamma}\right) \sigma_{t|t}^2.$$

   - update step: estimate $(m_{t|t}, \sigma_{t|t}^2)$, such that:

$$m_{t|t} = \frac{\sigma_w^2 m_{t|t-1}}{\sigma_w^2 + \sigma_{t|t-1}^2} + \frac{\sigma_{t|t-1}^2 y_t}{\sigma_w^2 + \sigma_{t|t-1}^2}, \qquad \sigma_{t|t}^2 = \frac{\sigma_w^2 \sigma_{t|t-1}^2}{\sigma_w^2 + \sigma_{t|t-1}^2}.$$

3. at $t = T$, repeat a prediction step to compute $(m_{T+1|T}, \sigma_{T+1|T}^2)$.

The computations of prediction step and update step are written in the functions `pred_step` and `update_step` respectively. Then, the main algorithm above is implemented by the function `gp.KF`, which includes the use of functions `pred_step` and `update_step` in STEP 2.

```
# prediction step
pred_step <- function(m.tt, sigma.tt, phi, sigma.v){

  # updated m_{t+1|t}
  m.t1t <- phi*m.tt
```

```
  # updated sigma_{t+1|t}
  sigma.t1t <- sqrt(sigma.v^2 + phi^2*sigma.tt^2)

  return(list(m=m.t1t, sigma=sigma.t1t))
}

# updating step
update_step <- function(m.t1t, sigma.t1t, phi, sigma.w, new.y){

  denominator <- sigma.w^2 + sigma.t1t^2

  # updated m_{t+1|t+1}
  m.t1t1 <- (sigma.w^2*m.t1t + sigma.t1t*new.y)/denominator

  # updated sigma_{t+1|t+1}
  sigma.t1t1 <- sqrt(sigma.w^2*sigma.t1t^2/denominator)

  return(list(m=m.t1t1, sigma=sigma.t1t1))

}

# KF inference
gp.KF <- function(y, gamma, sigma.w, m0, sigma0){

  # compute the parameters
  phi <- exp(-1/gamma)
  sigma.v <- sqrt(1- exp(-2/gamma))

  # length of y
  n <- length(y)

  # record parameters m_{t|t}, sigma_{t|t}, m{t+1|t}, sigma_{t+1|t}
  tt.record <- data.frame(t = NULL, m = NULL, sigma = NULL)
  t1t.record <- data.frame(t = NULL, m = NULL, sigma = NULL)

  # initialise
  m1 <- sigma0^2*(y[1]-m0)/(sigma0^2+sigma.w^2)
  sigma1 <- sqrt(sigma0^2 - sigma0^4/(sigma0^2+sigma.w^2))

  tt.record1 <- data.frame(t = 1, m = m1, sigma = sigma1)
  tt.record <- rbind(tt.record, tt.record1)

  for(t in 1:(n-1)){
    # prediction step
    pred.param <- pred_step(m1, sigma1, phi, sigma.v)
    m0 <- pred.param$m
    sigma0 <- pred.param$sigma
    t1t.record.i <- data.frame(t = t, m = m0, sigma = sigma0)
    t1t.record <- rbind(t1t.record, t1t.record.i)

    # updating step
    update.param <- update_step(m0, sigma0, phi, sigma.w, y[t+1])
    m1 <- update.param$m
```

```
    sigma1 <- update.param$sigma
    tt.record.i <- data.frame(t = t+1, m = m1, sigma = sigma1)
    tt.record <- rbind(tt.record, tt.record.i)
  }
  # prediction step at t=n
  pred.param <- pred_step(m1, sigma1, phi, sigma.v)
  m0 <- pred.param$m
  sigma0 <- pred.param$sigma
  t1t.record.n <- data.frame(t = n, m = m0, sigma = sigma0)
  t1t.record <- rbind(t1t.record, t1t.record.n)

  return(list(tt=tt.record, t1t=t1t.record))
}
```

Functions for smoothing and prediction under Kalman Filter

The smoothing task refers to the inference of $p(x_s|y_{1:T})$ where $s < T$. After obtaining $\left\{(m_{t|t}, \sigma^2_{t|t}), (m_{t+1|t}, \sigma^2_{t+1|t})\right\}_{t=1}^{T}$ from inference of GP using Kalman Filter, we can recursively compute $(m_{T-1|T}, \sigma^2_{T-1|T}), (m_{T-2|T}, \sigma^2_{T-2|T}), \ldots, (m_{1|T}, \sigma^2_{1|T})$: for $s = T-1, \ldots, 1$,

$$m_{s|T} = m_{s|s} + J_s(m_{s+1|T} - m_{s+1|s}), \qquad \sigma^2_{s|T} = \sigma^2_{s|s} + J_s^2(\sigma^2_{s+1|T} - \sigma^2_{s+1|s}),$$

where $J_s = \phi \frac{\sigma^2_{s|s}}{\sigma^2_{s+1|s}}$.

The above smoothing algorithm is implemented as function `smooth.KF` in the following code chunk.

```
# smoothing under KF
smooth.KF <- function(n, tt.record, t1t.record, alpha){
  # alpha: (1-alpha) credible interval

  # initialize at t=n
  smooth.record <-tt.record %>% filter(t==n)

  for(s in (n-1):1){
    ss.param <- tt.record %>% filter(t==s)
    s1s.param <- t1t.record %>% filter(t==s)
    s1n.param <- smooth.record %>% filter(t==(s+1))

    Js <- phi*(ss.param$sigma/s1s.param$sigma)^2

    m.sn <- ss.param$m + Js*(s1n.param$m - s1s.param$m)
    sigma2.sn <- ss.param$sigma^2 + Js^2*(s1n.param$sigma^2 - s1s.param$sigma^2)

    smooth.record.s <- data.frame(t = s, m = m.sn, sigma = sqrt(sigma2.sn))
    smooth.record <- rbind(smooth.record, smooth.record.s)

  }

  # (1-alpha)% credible interval
  smooth.record$upper <- qnorm(1-alpha/2, mean=smooth.record$m, sd=smooth.record$sigma)
  smooth.record$lower <- qnorm(alpha/2, mean=smooth.record$m, sd=smooth.record$sigma)
```

```
  return(smooth.record)
}
```

The prediction task is of the form $p(y_s|y_{1:T})$ with $s > T$ where $y_{1:T}$ is the observation values. Similarly, after obtaining $(m_{T|T}, \sigma^2_{T|T})$ from inference of GP using Kalman Filter, we can predict for the distribution of the next future state y_{T+1}, i.e. predict the parameters $(m_{T+1|T}, \sigma^2_{T+1|T})$, such that:

$$m_{T+1|T} = \phi m_{T|T}, \qquad \sigma^2_{T+1|T} = \sigma^2_v + \phi^2 \sigma^2_{T|T}.$$

The above prediction algorithm is implemented as function `pred.KF` in the following code chunk.

```
# prediction under KF
pred.KF <- function(m.nn, sd.nn, gamma){
  # m.nn: numeric, smoothed m_{n|n}
  # sd.nn: numeric, smoothed sd_{n|n}

  # compute the parameters
  phi <- exp(-1/gamma)
  sigma.v <- sqrt(1- exp(-2/gamma))

  # predicted m
  pred.m <- phi*last.smooth$m

  # predicted sd^2
  pred.sd2 <- last.smooth$sigma^2*phi^2 + sigma.v^2

  # output
  return(list(m = pred.m,
              sigma = sqrt(pred.sd2)))
}
```

Functions for hyperparameter selection

As parameters σ^2_v and ϕ can be written in terms of γ, we have in total of two hyper parameters (γ, σ^2_w) which need to be selected. As the marginal likelihood of $y_{1:t}$ can be written as follows:

$$p(y_{1:t}) = p(y_1) \prod_{k=2}^{t} p(y_k|y_{1:k-1}),$$

where $p(y_k|y_{1:k-1}) = \mathcal{N}(y_k; m_{k|k-1}, \sigma^2_{k|k-1} + \sigma^2_w)$, we can apply empirical Bayes approach to estimate MLE of $p(y_{1:T})$.

In the following code chunk, the function `compute_logp` is to return log-likelihood at given (γ, σ^2_w), and the function `optim_parm` is to find out the optimal pair of (γ, σ^2_w) which can maximize the negative log-likelihood value.

```
# compute log-likelihood
compute_logp <- function(y, gamma, sigma.w, m0, sigma0){
  # compute the KF inference
  KF.result <- gp.KF(y, gamma, sigma.w, m0, sigma0)
  t1t.record <- KF.result$t1t
```

```r
  # compute log-likelihood
  n <- length(y)
  y.mean <- t1t.record$m[-n]
  y.sd <- t1t.record$sigma[-n]+sigma.w^2
  value <- sum(log(dnorm(y[2:n], mean=y.mean, sd=y.sd))) +
    log(dnorm(y[1], mean = 0, sd = 1+sigma.w^2))

  return(value)

}

# compute optimal hyper-parameters
optim_parm <- function(y, m0, sigma0){
  opt_param <- optim(par = c(1,1),
                     fn = function(parm) -1*compute_logp(y, parm[1],
                                                         parm[2], m0, sigma0))
  return(list(gamma = opt_param$par[1],
              sigma.w=opt_param$par[2]))

}
```

Using Rcpp

To develop the computational performance of the above R code, we choose to interface R with C++ via Rcpp.

Here are functions `pred_step` and `update_step` written in Rcpp version:

```r
# prediction step in Rcpp version
sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;

// C++ function to perform the prediction step
List pred_step(double m_tt, double sigma_tt, double phi, double sigma_v) {
  double m_t1t = phi * m_tt;
  double sigma_t1t = sqrt(sigma_v * sigma_v + phi * phi * sigma_tt * sigma_tt);

  return List::create(Named("m") = m_t1t, Named("sigma") = sigma_t1t);
}

// Rcpp module to export the C++ function
RCPP_MODULE(MyModule) {
  function("pred_step", &pred_step, "Perform the prediction step");
}

// Define the entry point to the DLL
extern "C" void R_init_MyPackage(DllInfo* info) {
  R_registerRoutines(info, NULL, NULL, NULL, NULL);
  R_useDynamicSymbols(info, FALSE);
}')
```

```
# update step in Rcpp version
sourceCpp(code = '
  #include <Rcpp.h>
using namespace Rcpp;

// C++ function to perform the update step
List update_step(double m_t1t, double sigma_t1t, double phi, double sigma_w, double new_y) {
  double denominator = sigma_w * sigma_w + sigma_t1t * sigma_t1t;

  double m_t1t1 = (sigma_w * sigma_w * m_t1t + sigma_t1t * new_y) / denominator;
  double sigma_t1t1 = sqrt(sigma_w * sigma_w * sigma_t1t * sigma_t1t / denominator);

  return List::create(Named("m") = m_t1t1, Named("sigma") = sigma_t1t1);
}

// Rcpp module to export the C++ function
RCPP_MODULE(MyModule) {
  function("update_step", &update_step, "Perform the update step");
}

// Define the entry point to the DLL
extern "C" void R_init_MyPackage(DllInfo* info) {
  R_registerRoutines(info, NULL, NULL, NULL, NULL);
  R_useDynamicSymbols(info, FALSE);
}

')
```

Then, here is the modified function `gp.KF` to be consistent with the Rcpp functions `pred_step` and `update_step`:

```
gp.KF <- function(y, gamma, sigma.w, m0, sigma0){

  # compute the parameters
  phi <- exp(-1/gamma)
  sigma.v <- sqrt(1- exp(-2/gamma))

  # length of y
  n <- length(y)

  # record parameters m_{t|t}, sigma_{t|t}, m{t+1|t}, sigma_{t+1|t}
  tt.record <- data.frame(t = NULL, m = NULL, sigma = NULL)
  t1t.record <- data.frame(t = NULL, m = NULL, sigma = NULL)


  m1 <- sigma0^2*(y[1]-m0)/(sigma0^2+sigma.w^2)
  sigma1 <- sqrt(sigma0^2 - sigma0^4/(sigma0^2+sigma.w^2))

  tt.record1 <- data.frame(t = 1, m = m1, sigma = sigma1)
  tt.record <- rbind(tt.record, tt.record1)

  for(t in 1:(n-1)){
    # prediction step
```

```
    pred.param <- pred_step(m_tt = m1,
                            sigma_tt = sigma1,
                            phi = phi,
                            sigma_v = sigma.v)
    m0 <- pred.param$m
    sigma0 <- pred.param$sigma
    t1t.record.i <- data.frame(t = t, m = m0, sigma = sigma0)
    t1t.record <- rbind(t1t.record, t1t.record.i)

    # updating step
    update.param <-     update_step(m_t1t = m0,
                                    sigma_t1t = sigma0,
                                    phi = phi,
                                    sigma_w = sigma.w,
                                    new_y = y[t+1])
    m1 <- update.param$m
    sigma1 <- update.param$sigma
    tt.record.i <- data.frame(t = t+1, m = m1, sigma = sigma1)
    tt.record <- rbind(tt.record, tt.record.i)
  }
  # prediction step at t=n
  pred.param <- pred_step(m_tt = m1,
                          sigma_tt = sigma1,
                          phi = phi,
                          sigma_v = sigma.v)
  m0 <- pred.param$m
  sigma0 <- pred.param$sigma
  t1t.record.n <- data.frame(t = n, m = m0, sigma = sigma0)
  t1t.record <- rbind(t1t.record, t1t.record.n)

  return(list(tt=tt.record, t1t=t1t.record))
}
```

Here is the functions `pred.KF` written in Rcpp version:

```
sourceCpp(code='
#include <Rcpp.h>
using namespace Rcpp;

// C++ function to perform prediction in Kalman Filter
List predKF(double m_nn, double sd_nn, double gamma) {
  // Compute the parameters
  double phi = exp(-1 / gamma);
  double sigma_v = sqrt(1 - exp(-2 / gamma));

  // Predicted m
  double pred_m = phi * m_nn;

  // Predicted sd^2
  double pred_sd2 = pow(sd_nn, 2) * pow(phi, 2) + pow(sigma_v, 2);

  // Output
  return List::create(Named("m") = pred_m, Named("sigma") = sqrt(pred_sd2));
```

```
}

// Rcpp module to export the C++ function
RCPP_MODULE(MyModule) {
  function("predKF", &predKF, "Perform prediction in Kalman Filter");
}

// Define the entry point to the DLL
extern "C" void R_init_MyPackage(DllInfo* info) {
  R_registerRoutines(info, NULL, NULL, NULL, NULL);
  R_useDynamicSymbols(info, FALSE);
}
')
```

Summary

In this project, it is convenient to put these functions in separate files and call them for data analysis task.

Here is a list of files and their contained functions:

- R file `main_functions.R`: `gp.KF`,`compute_logp`, `optim_parm`
- Cpp file `pred_step.cpp`: `pred_step`
- Cpp file `update_step.cpp`: `update_step`
- Cpp file `predKF.cpp`: `predKF`

We can call these functions by implementing the following commands:

```
source("main_functions.R")
sourceCpp("pred_step.cpp")
sourceCpp("update_step.cpp")
sourceCpp("predKF.cpp")
```