GARMA DKR Revamp

Caden Hewlett

2025-07-09

Conditional Density of the Response: `dmeangamma`

Recall that the mean paramaterization of the conditional distribution of the gamma-distributed response variable y_t is given in terms of the shape α and mean μ_t as:

$$f_{Y_t}(y_t \mid \underline{\mathbf{H}}_t) = \frac{1}{\Gamma(\alpha)}\Big(\frac{\alpha}{\mu_t}\Big)^{\alpha} y_t^{\alpha-1} \exp\Big(-\frac{\alpha y_t}{\mu_t}\Big)$$

To implement this in `R`, we first compute the log-density;

$$\log\big(f_{Y_t}(y_t \mid \underline{\mathbf{H}}_t)\big) = \alpha\Big[\log(\alpha) + \log(y_t) - \log(\mu_t)\Big] - \log(y_t) - \log\big(\Gamma(\alpha)\big) - \frac{\alpha y_t}{\mu_t}$$

We can then write up an `R` function for this. The base condition for μ_t is similar to the base paramaterization in `dgamma`, wherein `scale = 1` by default.

Recall that we arrived at this paramaterization by letting the scale $\beta = \mu_t/\alpha$. Thus, to match the `dgamma` defaults where $\beta = 1$ we allow $1 = \mu_t/\alpha$ and hence $\mu_t = \alpha$ is the default functionality.

This should allow `dmeangamma(yt, shape) == dgamma(x, shape)` when only the shape is specified.

```
dmeangamma <- function(yt, shape, mu = shape, log = FALSE){
  # input validation
  if (any(shape < 0)) {
    stop("Shape Parameter must be Positive!")
  }
  if (any((mu / shape) <= 0)) {
    stop("Mean Parameter must be Positive!")
  }
  # compute log density
  log_density <- ( shape*(log(shape) + log(yt) - log(mu)) - log(yt)
            - lgamma(shape) - (shape*yt)/mu )

  # if yt not in supp(Yt)
  log_density[yt < 0] <- -1e10

  # return
  if(log == FALSE){
    return(exp(log_density))
  } else{
    return(log_density)
  }
}
# matches dgamma
```

```
alpha <- runif(1)
stopifnot(0 ==
            round(dgamma(1.8, shape = alpha) - dmeangamma(1.8, shape = alpha), 10))
```

GARMA(p,q) Function: Identity Link

Recall the form of the GARMA(p, q) model defined previously;

$$\eta_t = g(\mu_t) = \mathbf{x}_t^\top \boldsymbol{\beta} + \tau_t$$

Where

$$\tau_t = \sum_{j=1}^{p} \phi_j \big(g(y_{t-j}) - \mathbf{x}_{t-j}^\top \boldsymbol{\beta}\big) + \sum_{j=1}^{q} \theta_j \big(g(y_{t-j}) - \eta_{t-j}\big) \tag{1}$$

Now, we allow $g(u) = u$. We now model the mean directly since $\eta_t = \mu_t$, and the ARMA component τ_t is given by

$$\tau_t = \sum_{j=1}^{p} \phi_j (y_{t-j} - \tilde{y}_{t-j}) + \sum_{j=1}^{q} \theta_j (y_{t-j} - \mu_{t-j}), \quad \text{and} \quad \mu_t = \tilde{y}_t + \tau_t$$

Parameter Object

Recall our construction of $\boldsymbol{\Xi} = \{\mathbf{P}, \boldsymbol{\phi}, \boldsymbol{\theta}, \alpha\}$ which simplifies to $\boldsymbol{\Xi} = \{\boldsymbol{\beta}, \boldsymbol{\phi}, \boldsymbol{\theta}, \alpha\}$ in the general regression case.

The form of a parameter object $\boldsymbol{\Xi}$ is:

```
Xi_ex <- list(
  matrix(c(1, 2, 1.1, 0.9), nrow = 1), # k by 4 DKR parameter matrix
  c(0.1, 0.3), # AR parameters
  c(-0.4), # MA parameters
  2.1    # shape estimate
)
```

However, in order to pass to an optimizer we must present it in the form of a vector, i.e.

$$\boldsymbol{\Xi} = \Big\langle \underbrace{\beta_{01}, \beta_{11}, \delta_1, \sigma_1, \ldots, \sigma_k}_{4k \text{ Kernel Regression Terms}}, \underbrace{\phi_1, \ldots \phi_p}_{p \text{ AR Terms}}, \underbrace{\theta_1, \ldots \theta_q}_{q \text{ MA Terms}}, \underbrace{\alpha}_{\text{Shape}} \Big\rangle$$

We can quickly build a function to transform a vector from an optimizer into a parameter list;

```
par_to_xi <- function(P, k, p, q){
  # build empty list
  Xi <- vector("list", 4)
  # get regression parameters
  Xi[[1]] <- matrix(P[1:(4*k)], nrow = k, ncol = 4, byrow = TRUE)
  # check for AR params
  Xi[[2]] <- if (p > 0) P[(4*k + 1):(4*k + p)] else 0
  # ibid for MA params
  Xi[[3]] <- if (q > 0) P[(length(P) - q):(length(P) - 1)] else 0
  # then alpha is the last entry
  Xi[[4]] <- P[length(P)]
  # return list
  return(Xi)
}
```

This logic will be used in the prediction function later on. Further, the information object $\underline{\mathbf{H}}_t$ contains all previous response values, all covariates and the previous mean estimates. This is captured in the arguments of the state-space update equation.

State-Space Update Equation: `compute_mu`

We assume in the code to follow that `y[1]` $= y_1$.

```
compute_mu <- function(t, y, y_tilde, mu, Xi, g = I){

  # ---------------- #
  # --- AR Piece --- #
  # ---------------- #
  phi <- Xi[[2]]
  if (length(phi) > 0) {
    idx_ar <- t - seq_along(phi)
    term_ar <- ifelse(idx_ar > 0,
                      phi * (g(y[idx_ar]) - y_tilde[idx_ar]),
                      0)
  } else {
    term_ar <- 0
  }

  # ---------------- #
  # --- MA Piece --- #
  # ---------------- #
  theta <- Xi[[3]]
  if (length(theta) > 0) {
    idx_ma <- t - seq_along(theta)
    term_ma <- ifelse(idx_ma > 0,
                      theta * (g(y[idx_ma]) - g(mu[idx_ma])),
                      0)
  } else {
    term_ma <- 0
  }

  tau  <- sum(term_ar) + sum(term_ma)
  tau  <- ifelse(is.na(tau),  yes = 0, no = tau)
  mu   <- y_tilde[t] + tau

  return(mu)
}
```

Worked Test Example

We verify our function with a toy example.

Consider the time points $t \in \{1, 2, 3\}$. The observed streamflow for these time points is $y_t = \langle 10, 11, 12 \rangle$. Suppose our DKR function has fit $\tilde{y}_t = \langle 9, 10, 11 \rangle$. We are currently fitting a $(p, q) = (1, 1)$ model with $\hat{\phi}_1 = 0.5$ and $\hat{\theta}_1 = 0.3$.

Since $\tilde{y}_1 = 9$ is the first observation, $\tau_1 = 0$ and thus $\mu_1 = \tilde{y}_1 + \tau_1 = 9$. From this information, we can compute μ_2 as follows:

$$\begin{aligned}
\mu_2 &= \tilde{y}_2 + \tau_2 \\
&= \tilde{y}_2 + \sum_{j=1}^{1} \hat{\phi}_j (y_{2-j} - \tilde{y}_{2-j}) + \sum_{j=1}^{1} \hat{\theta}_j (y_{2-j} - \mu_{2-j}) \\
&= \tilde{y}_2 + \hat{\phi}_1 (y_1 - \tilde{y}_1) + \hat{\theta}_1 (y_1 - \mu_1) \\
&= 10 + (0.5)(10 - 9) + 0.3(10 - 9) \\
&= 10.8
\end{aligned}$$

We would expect the state space update function to behave in an identical manner.

```
y <- c(10, 11)
ytilde  <- c( 9, 10)
Xi <- list(NULL,  c(0.5),  c(0.3))
# compute first mu
mu_1 <- compute_mu(
  t = 1,
  y = y,
  y_tilde = ytilde,
  mu = NA,
  Xi = Xi
)
mu_2 <- compute_mu(
  t = 2,
  y = y,
  y_tilde = ytilde,
  mu = c(mu_1),
  Xi = Xi
)
test_that("toy example reproduces manual calculation", {
  expect_equal(mu_1,  9.0,  tolerance = 1e-12)
  expect_equal(mu_2, 10.8,  tolerance = 1e-12)
})
```

```
## Test passed
```

As the above implementation shows, each new μ_t relies on prior means up to lag $\max\{p, q\}$. Thus, we cannot linearly `apply` the state space update function to our computed $\tilde{y}_t$ values. Using a `for` loop is also inefficient for the scale of our data set (since in each iteration of the optimization we are computing 10,000 or more μ_ts, $\tilde{y}_t$s and so forth.)

C Implementation

We can translate the code to `C` to increase performance speed by orders of magnitude.

```
library(Rcpp)
Rcpp::cppFunction('
  Rcpp::NumericVector compute_mu_vec(const Rcpp::NumericVector& y,
                                     const Rcpp::NumericVector& y_tilde,
                                     const Rcpp::NumericVector& phi,
                                     const Rcpp::NumericVector& theta) {

    const int   n = y.size();
    const int   p = phi.size();
    const int   q = theta.size();
    Rcpp::NumericVector mu(n), tau(n);

    for (int t = 0; t < n; ++t) {

      // ------- AR part:
      double ar = 0.0;
      for (int j = 0; j < p; ++j) {
        int idx = t - j - 1;
        if (idx >= 0)
          ar += phi[j] * (y[idx] - y_tilde[idx]);
      }

      // ------- MA part:
      double ma = 0.0;
      for (int j = 0; j < q; ++j) {
        int idx = t - j - 1;
        if (idx >= 0)
          ma += theta[j] * (y[idx] - mu[idx]);
      }

      tau[t] = ar + ma;
      mu[t]  = y_tilde[t] + tau[t];
    }
    return mu;
  }')
```

We can verify this using our previously-verified baseline;

```
test_that("Rcpp example reproduces manual calculation", {
  expect_equal(compute_mu_vec(
    y = y,
    y_tilde = ytilde,
    phi = Xi[[2]],
    theta = Xi[[3]]
  ),
  c(9.0, 10.8),
  tolerance = 1e-12)
})
```

```
## Test passed
```

Worked Test Example

Further, we can check that it handles the $\max\{p, q\} \geq 2$ correctly by using our example, recalling $y_3 = 12$, $\tilde{y}_3 = 11$, $p = 2$ with $\hat{\phi}_2 = -0.15$, to find μ_3.

$$\begin{aligned}
\mu_3 &= \tilde{y}_3 + \tau_3 \\
&= \tilde{y}_3 + \sum_{j=1}^{2} \hat{\phi}_j (y_{3-j} - \tilde{y}_{3-j}) + \sum_{j=1}^{1} \hat{\theta}_j (y_{3-j} - \mu_{3-j}) \\
&= \tilde{y}_3 + \hat{\phi}_1 (y_2 - \tilde{y}_2) + \hat{\phi}_2 (y_1 - \tilde{y}_1) + \hat{\theta}_1 (y_2 - \mu_2) \\
&= 11 + 0.5(11 - 10) - 0.15(10 - 9) + 0.3(11 - 10.8) \\
&= 11.41
\end{aligned}$$

```
compute_mu_vec(
    y = c(10, 11, 12),
    y_tilde = c(9, 10, 11),
    phi = c(0.5, -0.15),
    theta = 0.3
)
```

```
## [1]  9.00 10.80 11.41
```

So, the Rcpp version is working for ARMA errors beyond $(1, 1)$.

Runtime Comparison Now, let's check the run time of the `Rcpp` version relative to the pure-`R` baseline for $n = 10,000$ random observations and $p = q = 1$. We will use `microbenchmark` to compare runtimes.

```
set.seed(1928)
n   <- 1e4
y   <- rnorm(n)
ytilde <- y + arima.sim(n, model = list(ar = c(0.5), ma = c(0.3)), sd = 0.1)
phi <- 0.5; theta <- 0.3

timing <- microbenchmark(
  Rcpp = compute_mu_vec(y, ytilde, phi, theta),
  R    = {
    mu <- numeric(n)
    for (t in seq_len(n))
      mu[t] <- compute_mu(t, y, ytilde, mu, list(NULL, phi, theta))
  },
  times = 10
)
```

The Rcpp implementation is exponentially faster than the pure-R alternative.

```
## Unit: microseconds
##  expr      min       lq      mean   median       uq      max neval cld
##  Rcpp    142.0    157.5    165.85    164.9    166.5    220.6    10  a
##     R 587272.7 608503.1 693408.70 676735.7 797332.5 833373.2    10   b
```

Response Prediction Function: `predict_response_garma`

In addition to storing the different parameter objects in Ξ, we implement a new list called `configs` which is intended to decrease the bulkiness of the `predict_response` function call.

Initially, we had to declare `p`, `kernel_type`, and `use_log`. In the Gamma-GARMA version, we also have `q` and potentially `g` to pass to the function.

Further, `k` was originally inferred from the length of the `params` object, since the parameter object only contained DKR kernel parameters and AR parameters. However, the inference is much less straightforward now (as we saw with the `par_to_xi` function) and hence `k` must also be provided.

The `configs` list seeks to remedy this problem. The default construction of this object is a named list of the following structure.

```
configs_default <- list(
  k = 1, # number of windows
  p = 0, # AR order
  q = 0, # MA order
  g = I, # identity link
  kernel_type = "gamma",
  use_log = FALSE # log-transform y and y_tilde
)
```

This change allows the function signature to be dramatically simplified.

```
library(DKR)
predict_response_garma <- function(xt, zt, yt, params, configs) {
  # default config is no AR, no MA
  configs_default <- list(
    k = max(1, length(params)/4 - 1),
    p = 0,  q = 0, g = I,
    kernel_type = "gamma",
    use_log = FALSE
  )
  # modify by user configuration
  configs <- modifyList(configs_default, configs)
  # then extract relevant hyper-parameters
  k <- configs$k
  p <- configs$p
  q <- configs$q
  # ----------- #
  # --- DKR --- #
  # ----------- #
  pars <- as.data.frame(matrix(params[1:(4*k)], nrow = k, ncol = 4, byrow = TRUE))
  colnames(pars) <- c("beta0", "beta1", "logdelta", "logsigma")
  # build kernels for each set of parameters
  kernels <- sapply(1:k,
    function(i) build_kernel(pars$logdelta[i], pars$logsigma[i],
                             type = configs$kernel_type)
  )
  # convert kernel to list if there is only one kernel
  if (k == 1)
    kernels <- list(kernels)
  # zt convolution
```

```r
  zt_conv <- sapply(1:k, function(i)
    convolve_kernel(zt, kernels[[i]]), simplify = "matrix")
  # compute beta for each kernel
  beta_z_conv    <- sweep(zt_conv, 2, pars$beta1, `*`)
  beta <- beta_z_conv + matrix(
    pars$beta0,
    nrow = nrow(beta_z_conv),
    ncol = length(pars$beta0),
    byrow = TRUE
  )
  # convolve xt with kernels
  xt_conv <- sapply(
    1:k,
    function(i) convolve_kernel(xt, kernels[[i]]),
    simplify = "matrix"
  )
  # compute kernel contributions
  yt_tilde       <- rowSums(xt_conv * beta)
  # ------------------ #
  # --- ARMA Error --- #
  # ------------------ #
  # check for AR params
  phi <- if (p > 0) params[(4*k + 1):(4*k + p)] else 0
  # ibid for MA params
  theta <- if (q > 0) params[(length(params) - q):(length(params) - 1)] else 0
  # then alpha is the last entry (used in log-likelihood)
  alpha <- params[length(params)]
  # use our mu_t computation function
  mu_t <- compute_mu_vec(y = yt, y_tilde = yt_tilde, phi, theta)
  # return results
  list(
    mu_hat         = mu_t,
    yt_tilde       = yt_tilde,
    tau_t          = mu_t - yt_tilde,
    beta           = beta
  )
}
```

Manual Computation, End-to-End

Now that we have a complete model prediction function, we must manually verify it.

For our toy example, suppose that we have a simple one Gamma Kernel DKR model with ARMA$(1,1)$ errors. The parameters of this model are given by:

$$\begin{aligned}\mathbf{P} &= \begin{bmatrix}\beta_0, \beta_1, \delta, \sigma\end{bmatrix} = \begin{bmatrix}0.5, -0.15, 1, 0.25\end{bmatrix} \\ \boldsymbol{\phi} &= \langle \phi_1 \rangle = \langle 0.45 \rangle \\ \boldsymbol{\theta} &= \langle \theta_1 \rangle = \langle 0.3 \rangle\end{aligned}$$

Thus, the kernel κ is given by

$$\kappa_i = \frac{1}{\Gamma(\alpha)}\Big(\Gamma(\alpha, i/\theta) - \Gamma(\alpha, (i+1)/\theta)\Big) = \frac{1}{\Gamma(1)}\Big(\Gamma(1, 4i) - \Gamma(1, 4(i+1))\Big)$$

Further, the lag range for shape α and rate $\lambda = 1/\theta$ is given by

$$\ell \in \left[\left\lfloor \frac{\gamma^{-1}(\alpha, 10^{-3})}{\lambda} \right\rfloor, \left\lceil \frac{\gamma^{-1}(\alpha, 1-10^{-3})}{\lambda} \right\rceil\right]$$

Thus, the general equation for our discretized gamma kernel would be:

$$\kappa_i = \left\{ \frac{1}{\Gamma(\delta_i)}\Big(\Gamma(\delta_i, \ell/\sigma_i) - \Gamma(\delta_i, (\ell+1)/\sigma_i)\Big) \mid \ell \in \left[\left\lfloor \gamma^{-1}(\delta_i, 10^{-3})\sigma_i \right\rfloor, \left\lceil \gamma^{-1}(\delta_i, 1-10^{-3})\sigma_i \right\rceil\right] \right\}$$

Where the range for ℓ is computed via Best and Roberts (1975, source) gamma quantile algorithm used in `qgamma`, where $\gamma^{-1}(\dots)$ is the inverse lower incomplete gamma function.

The following computes this algorithm in Python.

```
import math
from scipy.special import gammaincinv
# gamma quantile
def gamma_quantile_function(shape, rate, p):
    return gammaincinv(shape, p) / rate
# example
range(
  math.floor(gamma_quantile_function(1, 1/0.25, 0.001)),
  math.ceil(gamma_quantile_function(1, 1/0.25, 1-0.001))
)
```

```
## range(0, 2)
```

As we see from the above, $\ell \in [0, 2]$ in our parameter case. Thus the discretized kernel is given by

$$\kappa = \Big\{\Gamma(1, 4\ell) - \Gamma(1, 4(\ell+1)) \mid \ell \in [0, 2]\Big\}$$

Which we can compute as follows:

```
library(expint)
sapply(0:(2-1), function(ell){
  gammainc(1, 4*ell)-gammainc(1, 4*(1 + ell))
})
```

```
## [1] 0.98168436 0.01798018
```

```
DKR::build_kernel(log(1), log(0.25), "gamma")
```

```
## [1] 0.98201379 0.01798621
```

Notice that in order to match our current DKR code, we have $\ell \in [0, 2)$. It's really a manner of semantics where we consider the last lag to be.

Now that we have computed the kernel, we let our data are given by

$$\begin{bmatrix} \mathbf{y}_T & \mathbf{x}_T & \mathbf{z}_T \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 3 & 2 & -1 \\ 4 & 3 & 1 \end{bmatrix}$$

Suppose we wish to estimate μ_T in our complete construction.

We first compute $\tilde{y}_T$, recalling that $k = 1$ hence the regression component for $t = 1$ is given by:

$$\tilde{y}_1 = (\beta_0 + \beta_1(z_1\kappa_1))(x_1\kappa_1) = (0.50 - 0.15(1 \cdot 0.982))(1 \cdot 0.982) = 0.346$$

Similarly,

$$\begin{aligned} \tilde{y}_2 &= (\beta_0 + \beta_1(z_2\kappa_1 + z_1\kappa_2))(x_2\kappa_1 + x_1\kappa_2) \\ &= (0.50 - 0.15(-1 \cdot 0.982 + 1 \cdot 0.018))(2 \cdot 0.982 + 1 \cdot 0.018) \\ &= 1.277 \end{aligned}$$

And, recalling $\text{length}(\kappa) = 2$,

$$\begin{aligned} \tilde{y}_3 &= (\beta_0 + \beta_1(z_3\kappa_1 + z_2\kappa_2))(x_3\kappa_1 + x_2\kappa_2) \\ &= (0.50 - 0.15(1 \cdot 0.982 - 1 \cdot 0.018))(3 \cdot 0.982 + 2 \cdot 0.018) \\ &= 1.0598 \end{aligned}$$

The manual computation correctly aligns with both the original `DKR::predict_target` and the new implementation:

```
##          [,1]     [,2]     [,3]
## Old 0.3463542 1.277614 1.059795
## New 0.3463542 1.277614 1.059795
```

Now, we compute τ_t and μ_t for each iterate, recalling $\phi = 0.45$ and $\theta = 0.3$. By construction, $\tau_1 = 0$.

$$\begin{aligned} \tau_2 &= \phi(y_1 - \tilde{y}_1) + \theta(y_1 - \mu_1) = 0.45(2 - 0.346) + 0.3(2 - 0.346) = 1.2405 \\ \mu_2 &= \tilde{y}_2 + \tau_2 = 1.2776 + 1.2405 = \boxed{2.5181} \end{aligned}$$

Finally, μ_3 can be found via the following (noticing the MA component in effect)

$$\begin{aligned} \tau_3 &= 0.45(3 - 1.2776) + 0.3(3 - 2.5181) = 0.91965 \\ \mu_3 &= \tilde{y}_3 + \tau_3 = 1.0598 + 0.9197 = \boxed{1.9795} \end{aligned}$$

Thus, our manually computed μ values are $\mu_T = \{0.346, 2.5181, 1.9795\}$. We can now *finally* verify our `predict_response_garma` function.

```
library(DKR)
computed <- predict_response_garma(
  xt = x,
  zt = z,
```

```
  yt = y,
  params = c(params, 0.45, 0.30, 1),
  configs = list(k=1, p=1, q=1)
)
round(computed$mu_hat, 4)
```

```
## [1] 0.3464 2.5178 1.9795
```

Everything matches up as expected (with a bit of rounding error)!

Log-Likelihood and Error Function

The log-likelihood of this parameter set can then be found with our previously-defined `dmeangamma` function and $\hat{\alpha} = 1$ for simplicity.

```
sum(dmeangamma(yt = y, shape = 1, mu = computed$mu_hat, log = TRUE))
```

```
## [1] -9.532589
```

Thus, for our optimization, we can construct the `error` function as

```
error <- function(xt, zt, yt, params, configs) {
  # predict response using object parameters
  preds <- predict_response_garma(xt, zt, yt, params, configs)
  # check for infinite or NA fits
  if (any(!is.finite(preds$mu_hat)) | any(is.na(preds$mu_hat))) {
    return(-1e10)
  } else{
    # otherwise return the log-likelihood using our density function
    sum(dmeangamma(
      yt = yt,
      shape = params[length(params)], # alpha is always the last entry of Xi
      mu = preds$mu_hat,
      log = TRUE
    ))
  }
}
error(x,z,y,  params = c(params, 0.45, 0.30, 1),
  configs = list(k=1, p=1, q=1))
```

```
## [1] -9.532589
```