Adding new distributions in hmmTMB

Théo Michelot

2024-04-30

The package hmmTMB includes many possible distributions for the observation model:

```
names(hmmTMB:::dist_list)
```

```
 [1] "beta"       "binom"      "cat"        "dir"        "exp"
 [6] "foldednorm" "gamma"      "gamma2"     "lnorm"      "mvnorm"
[11] "nbinom"     "norm"       "pois"       "t"          "truncnorm"
[16] "tweedie"    "vm"         "weibull"    "wrpcauchy"  "zibinom"
[21] "zigamma"    "zigamma2"   "zinbinom"   "zipois"     "zoibeta"
[26] "ztnbinom"   "ztpois"
```

This vignette provides a technical description of the steps required to add a new distribution to the package. If the distribution you want is not implemented, you can contact us about it; of course, you can also fork the Github repository, add the distribution, and submit a pull request.

1 R code

1.1 Create a `Dist` object (`R/dist_def.R`)

In hmmTMB, a distribution is an object of the `Dist` class, which stores information such as the number of parameters, probability density function, and link functions. To add a new distribution, a new `Dist` object must be created using `Dist$new()` in the file `R/dist_def.R`. The chunk below shows the code required to create an object for the exponential distribution.

```
dist_norm <- Dist$new(
  name = "norm",
  name_long = "normal",
  pdf = dnorm,
```

```
  cdf = pnorm,
  rng = rnorm,
  link = list(mean = identity, sd = log),
  invlink = list(mean = identity, sd = exp),
  npar = 2,
  parnames = c("mean", "sd"),
  parapprox = function(x) {
    return(c(mean(x), sd(x)))
  }
)
```

Our convention is to name the `Dist` object as `dist_name` where `name` is the name of the distribution. The arguments passed to `Dist$new()` are the following.

- `name`: Name of the distribution, which will be used to define the observation model.
- `name_long`: Expanded form of the name of the distribution (optional).
- `pdf`: A function object for the probability density function (or probability mass function). If there is a built-in R function, like `dnorm`, this can be used directly; otherwise, a custom function must be written. This should take as inputs `x` (the value at which to evaluate the pdf), the parameters of the function, and `log` (`TRUE`/`FALSE` to indicate whether the log-pdf is required). It should be vectorised with respect to `x`; that is, if `x` is a vector of length `n`, then this function outputs the vector of `n` evaluations of the pdf at those values.
- `cdf`: A function object for the cumulative distribution function, with similar arguments as the pdf (except `log`).
- `rng`: A function object to generate random values, with inputs `n` (the number of random values) and the distribution parameters.
- `link`: A named list of function objects, with one entry for the link function of each distribution parameter. Common choices include `identity` ($\theta \in \mathbb{R}$), `log` ($\theta > 0$), and `qlogis` ($0 < \theta < 1$).
- `invlink`: A named list of function objects, with one entry for the inverse link function of each distribution parameter. These should be the inverse functions of the entries of `link`. Common choices include `identity` ($\theta \in \mathbb{R}$), `exp` ($\theta > 0$), and `plogis` ($0 < \theta < 1$).
- `npar`: Number of parameters.
- `parnames`: Names of parameters. This should match the inputs of `pdf`, `cdf` and `rng`, and the names of the elements of `link` and `invlink`.
- `parapprox`: A function object that takes a vector of observations, and returns estimates for the parameters of the distribution. This is only used to suggest initial parameter

values in `Observation$suggest_initial()`, and can be defined as a function that returns `NA` if there is no simple heuristic to return point estimates.

1.2 Add distribution to `dist_list` (`R/dist_def.R`)

All `Dist` objects are stored in a list named `dist_list`, which is defined at the bottom of the file `R/dist_def.R`. The new distribution should be added to this list as an entry of the form `name = dist_name`. As we will see in Section 2.2, the order of distributions in this list matters to link the R and C++ sides of the code, so the simplest option is to add the new distribution to the end of the list.

```
dist_list <- list(beta = dist_beta,
                  binom = dist_binom,
                  cat = dist_cat,
                  dir = dist_dir,
                  exp = dist_exp,
                  ...
                  zipois = dist_zipois,
                  zoibeta = dist_zoibeta,
                  ztnbinom = dist_ztnbinom,
                  ztpois = dist_ztpois)
```

2 C++ code

2.1 Create a distribution class (`src/dist_def.hpp`)

On the C++ side, distributions are defined as classes that inherit from a parent class named `Dist`. A new such class must be defined for the new distribution. The C++ distribution has fewer elements than its R counterpart because only the log-likelihood function is implemented in C++ (and the log-likelihood doesn't need to know the cumulative distribution function, or to be able to generate random numbers, for example). The new class must be added to the file `src/dist_def.hpp`, and the chunk below shows what the format should be.

```
template<class Type>
class Normal : public Dist<Type> {
public:
  // Constructor
  Normal() {};
  // Link function
```

```
  vector<Type> link(const vector<Type>& par, const int& n_states) {
    vector<Type> wpar(par.size());
    // mean
    for (int i = 0; i < n_states; ++i) wpar(i) = par(i);
    // sd
    for (int i = n_states; i < 2 * n_states; ++i) wpar(i) = log(par(i));
    return(wpar);
  }
  // Inverse link function
  matrix<Type> invlink(const vector<Type>& wpar, const int& n_states) {
    int n_par = wpar.size()/n_states;
    matrix<Type> par(n_states, n_par);
    // mean
    for (int i = 0; i < n_states; ++i) par(i, 0) = wpar(i);
    // sd
    for (int i = 0; i < n_states; ++i) par(i, 1) = exp(wpar(i + n_states));
    return(par);
  }
  // Probability density/mass function
  Type pdf(const Type& x, const vector<Type>& par, const bool& logpdf) {
    Type val = dnorm(x, par(0), par(1), logpdf);
    return(val);
  }
};
```

The class should be given an informative name (`Normal` in the example), which doesn't have to match the name chosen in R. Then, the following four functions need to be implemented.

- The name of the constructor (e.g., `Normal`) should be updated to match the name of the new distribution.
- The `link` function takes as inputs a vector of parameters on the natural scale and the number of HMM states, and returns a vector of parameters on the link scale. In the normal example, the input is a vector with `n_states` means (identity link) and `n_states` standard deviations (log link).
- The `invlink` function takes as inputs a vector of parameters on the link scale and the number of HMM states, and returns a matrix of parameters on the natural scale. The matrix should have one row for each state and one column for each parameter.
- The `pdf` function takes as inputs the point `x` at which the pdf should be evaluated, a

vector of parameters `par` on the natural scale, and a logical variable indicating whether the log-pdf should be returned. In the example, we directly use the `dnorm` function implemented in TMB, but this might require custom code if the pdf is not standard.

The names of these functions and the lists of arguments should not be modified; only their content should be changed for the new distribution.

2.2 Add distribution to list (`src/dist.hpp`)

There needs to be a way to link the distribution on the R side to its C++ implementation. For this purpose, an integer code is automatically generated in `dist_def.R` for each distribution, starting at 0 and in the order defined in the `dist_list` object (see Section 1.2). In C++, each distribution is associated with its code in the file `src/dist.hpp` in a `switch` statement. The new distribution should be added following the same format as other distributions, simply changing the number after `case` and the naem of the distribution after `new`. The code should match the R code, so this list needs to be in the same order as the R object `dist_list`. If you added the distribution to the end of `dist_list` in Section 1.2, you can add the distribution to the end of the `switch` statement here.

```
template <class Type>
std::unique_ptr<Dist<Type>> dist_generator(const int& code) {
  switch (code) {
  case 0:
    return(std::unique_ptr<Dist<Type>>(new Beta<Type>));
  case 1:
    return(std::unique_ptr<Dist<Type>>(new Binomial<Type>));
  case 2:
    return(std::unique_ptr<Dist<Type>>(new Categorical<Type>));
  case 3:
    return(std::unique_ptr<Dist<Type>>(new Dirichlet<Type>));
  case 4:
    return(std::unique_ptr<Dist<Type>>(new Exponential<Type>));
  ...
  case 23:
    return(std::unique_ptr<Dist<Type>>(new ZeroInflatedPoisson<Type>));
  case 24:
    return(std::unique_ptr<Dist<Type>>(new ZeroOneInflatedBeta<Type>));
  case 25:
    return(std::unique_ptr<Dist<Type>>(new ZeroTruncatedNegativeBinomial<Type>));
```

```
  case 26:
    return(std::unique_ptr<Dist<Type>>(new ZeroTruncatedPoisson<Type>));
  default:
    return(std::unique_ptr<Dist<Type>>(new Normal<Type>));
  }
}
```

3 Test in a simulation study

The best way to check that all components of the new distribution were implemented correctly is to run a simulation. That is, you can generate data from an HMM, and check whether hmmTMB can recover the model parameters. (This might not detect some errors, but is a good sanity check anyway.) The code below demonstrates what such a simulation could look like, for the distribution `"norm"`. A more extensive simulation study (over many simulated data sets) could be used to investigate the performance of the model in more detail.

```
####################
## Simulate data ##
####################
n <- 1e4

# Simulate state process (2-state Markov chain)
tpm <- matrix(c(0.95, 0.1, 0.05, 0.9), 2, 2)
S <- rep(NA, n)
S[1] <- 1
for(i in 2:n) {
  S[i] <- sample(1:2, size = 1, prob = tpm[S[i-1],])
}

# Simulate observation process (normal distributions)
means <- c(1, 3)
sds <- c(1, 2)
Z <- rnorm(n = n, mean = means[S], sd = sds[S])

# Data frame for hmmTMB
data <- data.frame(Z = Z)
```

```r
###############
## Fit model ##
###############
library(hmmTMB)

# State model
hid <- MarkovChain$new(data = data, n_states = 2)

# Observation model
par0 <- list(Z = list(mean = c(0, 4), sd = c(0.5, 2)))
obs <- Observation$new(data = data, dists = list(Z = "norm"), par = par0)

# Create and fit HMM
hmm <- HMM$new(obs = obs, hid = hid)
hmm$fit(silent = TRUE)

# Check that parameters were recovered
lapply(hmm$par(), round, 3)
```

```
$obspar
, , 1

       state 1 state 2
Z.mean   0.991   3.000
Z.sd     0.996   1.983


$tpm
, , 1

        state 1 state 2
state 1   0.943   0.057
state 2   0.111   0.889
```