Package 'imabc'

April 12, 2021

Type Package

Title Incremental Mixture Approximate Bayesian Computation (IMABC)

Version 1.0.0

Description Provides functionality to perform a likelihood-
free method for estimating the parameters of complex models
that results in a simulated sample from the posterior distribution of model parameters given targets. The method begins
with a accept/reject approximate bayes computation (ABC) step applied to a sample of points from the prior distribution
of model parameters. Accepted points result in model predictions that are within the initially specified tolerance
intervals around the target points. The sample is iteratively updated by drawing additional points from a mixture of
multivariate normal distributions, accepting points within tolerance intervals. As the algorithm proceeds, the
acceptance intervals are narrowed. The algorithm returns a set of points and sampling weights that account for the
adaptive sampling scheme. For more details see Rutter, Ozik, DeYoreo, and Collier (2018) <arXiv:1804.02090>.

License GPL-3

URL https://github.com/carolyner/imabc

BugReports https://github.com/carolyner/imabc/issues

Depends R (>= 3.2.0)

Imports MASS, data.table, foreach, parallel, truncnorm, lhs, methods,
stats, utils

Encoding UTF-8

RoxygenNote 7.1.1

NeedsCompilation no

Author Christopher, E. Maerzluft [aut, cre],
Carolyn Rutter [aut, cph] (<https://orcid.org/0000-0002-4396-8594>),
Jonathan Ozik [aut] (<https://orcid.org/0000-0002-3495-6735>),
Nicholson Collier [aut] (<https://orcid.org/0000-0002-2376-4156>)

Maintainer ``Christopher, E. Maerzluft'' <cmaerzlu@rand.org>

Repository CRAN

Date/Publication 2021-04-12 08:30:03 UTC

R topics documented:

`define_target_function`
Define Target Function(s)

Description

Helps the user build a target function that applies parameters to a function or set of functions. The results of this function are then compared to the target goals to determine the goodness of fit of the parameters.

Usage

```
define_target_function(targets, priors, FUN = NULL, use_seed = FALSE)
```

Arguments

`targets`	targets object. Created using the define_targets function. Available to use within the target function(s) See Details.
`priors`	priors object. Created using the define_priors function. Available to use within the target function(s) See Details.
`FUN`	Optional function. If the user does not define target functions using define_targets, they can specify a single function here. See Details.
`use_seed`	logical. Should the algorithm set a seed before each set of parameters is sent to the target function(s). The seed is set once for each set of parameters.

Details

FUN: While the user can define a function for each target they create using add_target, there may be times when the user wants to have more control over how the functions are evaluated. For example, one target may be a function of another target and a parameter. If the target functions are created using define_targets, the first function would have to be evaluated twice. However, by using FUN,

the user can create a function where 'T1 = f(x1)' and 'T2 = g(T1, x2)'. This is especially helpful if the target functions take a long time to run.

Specifying Parameters as Target Function Inputs: Whether specifying target functions individually or through the FUN input, the inputs must follow a certain set of rules in order to be applied correctly. It is important to remember that the input(s) are ultimately based on the values specified in the priors object. Thus the target function(s) inputs will have to reference those parameters. This can be done either as a single vector of values (e.g. 'function(x) ... '), or individual inputs for each parameter (e.g. 'function(x1, x2) ... '). If a single vector is used, all parameters will be passed to the target function as a named vector and the user can reference that vector by either using the parameter names (e.g. 'x["x1"]') or by using the order a parameter was added in define_priors as the index number (e.g. 'x[1]'). If individual inputs are given for each parameter, then only the ones specified as inputs will be sent to the given target function. If this route is taken the inputs into the target function must match the name(s) of the parameters being used. The single vector method will be most useful when creating a single target function with FUN, while the individual inputs method is nice for simple target functions added via defined_targets.

Special Target Function Inputs: Beyond specifying the parameters, the user may optionally choose to include the targets and priors objects as inputs into the target function(s). These inputs must be specified as targets and priors respectively (e.g. 'function(x, targets, priors)'). They give you access to all the values defined and updated over the course of a calibration. When using FUN, this can be especially useful if one target calculation is extremely fast while another one is extremely slow; the user can simulate the first, check it against the target bounds, and choose to sidestep the slower target function in order to get a speed boost for the entire calibration. To see what values are available for each object use the names function on a recently created object for each class.

Target Function Outputs: While define_targets will handle the outputs appropriately for imabc, the user must be careful to do the same when specifying a target function through FUN. It is required that the results of FUN is a vector whose length is equal to the number of targets defined. If the vector is named, imabc will ensure that the order of the vector is correct before performing any calculations with the results. The names must match the names given to targets in define_targets; if you didn't provide names, you can find the generated names using 'attr(targets, "target_names")'. If the vector is not named, the order of the results in the vector must match the order the targets were added in define_targets.

Value

An imabc ready function.

Examples

```
priors <- define_priors(
  x1 = add_prior(dist_base_name = "unif"),
  x2 = add_prior(density_fn = "dnorm", mean = 0.5, sd = 0.25)
)
targets <- define_targets(
 T1 = add_target(target = 0.5, starting_range = c(0.2, 0.9), stopping_range = c(0.48, 0.51)),
  add_target(
    target_name = "T2",
    target = 1.5, starting_range = c(1.0, 2.0), stopping_range = c(1.49, 1.51)
  )
)
```

```
fn1 <- function(x1, x2) { x1 + x2 + sample(c(-1, 1), 1)*rnorm(1, 0, 0.1) }
fn2 <- function(x1, x2) { x1 * x2 + sample(c(-1, 1), 1)*rnorm(1, 0, 0.1) }
fn <- function(x1, x2) {
  res <- c()
  res["T2"] <- fn2(x1, x2)
  res["T1"] <- fn1(x1, x2)
  return(res)
}
target_fun <- define_target_function(targets, priors, FUN = fn, use_seed = FALSE)
```

---

imabc *Incremental Mixture Approximate Bayesian Computation (IMABC)*

Description

Calibrates a model using the IMABC algorithm.

Usage

```
imabc(
  target_fun,
  priors = NULL,
  targets = NULL,
  N_start = 1,
  N_centers = 1,
  Center_n = 50,
  N_cov_points = 0,
  N_post = 100,
  sample_inflate = 1.5,
  max_iter = 1000,
  seed = NULL,
  latinHypercube = TRUE,
  backend_fun = NULL,
  output_directory = NULL,
  output_tag = "timestamp",
  previous_results_dir = NULL,
  previous_results_tag = NULL,
  verbose = TRUE,
  validate_run = TRUE
)
```

Arguments

target_fun	A function that generate target values given parameters (i.e., 'the model'). The use of define_target_function is stronlgy advised to ensure that the function takes in the correct values and correctly returns results.

priors — A priors object created using define_priors. This contains information regarding the parameters that are being calibrated. Is ignored if starting from previous results.

targets — A targets object created using define_targets. This contains information regarding the target values which will be used to evaluate simulated parameters. Is ignored if starting from previous results.

N_start — numeric(1). The number of draws to simulate for the first iteration.

N_centers — numeric(1). The number of centers to use for exploring the parameter space.

Center_n — numeric(1). The number of points to add around each center

N_cov_points — numeric(1). The minimum number of points used to estimate the covariance matrix of valid parameters nearest each center point. The covariance matrix is used when simulating new parameter draws around the center. If 0 (default), uses 25*number of parameters.

N_post — numeric(1). The weighted sample size that must be achieved using valid parameter values in order to stop algorithm.

sample_inflate — numeric(1). When generating new results for a given center, how many additional samples should be simulated to ensure enough valid (within range) parameters draws are simulated for the center.

max_iter — numeric(1). The maximum number of iterations to attempt.

seed — numeric(1). The seed to set for reproducibility.

latinHypercube — logical(1). Should algorithm use a Latin Hypercube to generate first set of parameters.

backend_fun — function. For advanced users only. Lets to user evaluate the target function(s) using their own backend, i.e., simulate targets with an alternative parallel method. Only necessary if the backend method is not compatible with foreach. See details for requirements.

output_directory — character(1). Path to save results to. If NULL (default), no results are saved. If a path is provided results are saved/updated every iteration. See details for more information.

output_tag — character(1). Tag to add to result files names. "timestamp" (default) is a special code that adds the time and date the code was executed.

previous_results_dir — Optional character(1). Path to results stored during a previous run. If the user wishes to restart a run that didn't complete the calibration, they can continue by using the outputs stored during the previous run.

previous_results_tag — Optional character(1). The tag that was added to the previous run output files.

verbose — logical(1). Prints out progress messages and additional information as the model works.

validate_run — logical(1). If this is TRUE and an output_directory is specified, the function will save all parameters generated by the model - even ones that were deemed invalid based on their simulated targets.

Details

The user specifies the calibrated parameters, their prior distributions, calibration targets with initial and final acceptance intervals, and the function (i.e., the model) used to generate targets given calibrated parameters The algorithm begins by drawing a sample of vectors from the parameter space based on prior distributions. This initial sample can be drawn using a Latin hypercube. The algorithm identifies and retains parameter vectors that result in generated targets that are within the current acceptance intervals. The algorithm iteratively updates this sample and narrows the acceptance intervals until either 1) the algorithm reaches the final acceptance intervals around each target and identifies the requested sample of parameter vectors that generate targets within these acceptance intervals, or the algorithm completes the maximum number of iterations. The algorithm can be restarted to continue iterating.

A technical description of the imabc algorithm is provided in Rutter CM, Ozik J, DeYoreo M, Collier N. Microsimulation model calibration using incremental mixture approximate Bayesian computation. Ann. Appl. Stat. 13 (2019), no. 4, 2189-2212. doi:10.1214/19-AOAS1279. https://projecteuclid.org/euclid.aoas/1574910041.

The imabc package implements a small modification to the approach described in the 2019 AOAS paper. In the imabc package, the user specifies initial and final acceptance intervals directly. This approach is more flexible than the approach described in the paper and more easily incorporates asymmetric acceptance intervals.

N_cov_points relation to the number of parameters::

When the algorithm has enough quality draws, it estimates the covariance between parameters and uses these relations in order to improve future simulations of parameters. However, this can only work if the covariance matrix is not singular. When a covariance matrix is singular, imabc will replace it with an independent covariance matrix (a diagonal matrix of the variances of the parameters) to avoid any calculation errors. Setting N_cov_points to be less than the number of parameters will lead to singularness in a covariance matrix. The algorithm can still run but will be not as efficient or may not be able to calibrate completely.

Custom Backend Function::

The primary run handler takes each row from the simulated draws and provides the appropriate information to the target_fun function as inputs. This includes pulling the parameter values as a named vector, pulling a unique seed generated for each set of parameters, as well as passing the current priors and targets objects. This is done using the foreach function from the foreach package. This allows the user to register their own preferred parallel backend before running the imabc function so long as it is compatible with foreach. If the user does not provide a parallel backend, foreach will run the analysis in sequence by default and provide a warning indicating such the first time the imabc function is run within a session.

However, since not all parallel backends are compatible with this method, we have provided a way for the user to add their own run handling method. To utilize this feature, the user must create a function that meets a couple requirements in order to work properly.

The first requirement is that the backend function have inputs in the following order: the data.table of all parameters to be evaluated, the names of all the parameters being calibrated, the target function to be used for evaluating parameters, a list that includes the priors object and the targets object. The user can name these inputs whatever they prefer but the correct order and number of inputs will be expected (i.e. the user must create a function with four inputs, the first will be the parameter data.table, and so on.). The user can utilize any piece of info passed to these parameters

as well. This includes unique seed values passed as a column of the parameter data.table (called "seed"), and the current targets and priors objects passed in the fourth input. The priors and targets objects are named priors and targets respectively in the fourth input list.

The last requirement is that the returned object be a data.table of simulated target values. Each row represents a set of results from the target_fun for a given set of parameters and each column represents a target value based on the targets object. If the final output of the custom backend returns a data.table with column names identical to the target names, the order of the columns will be verified by imabc. If the final output of the backend does not include column names that match the target names, the user must ensure that they are in the same order as the targets object. If they are not in the appropriate order, information may be attached to the wrong target and lead to errors.

Do not use the custom backend unless you are confident you understand what is expected of the run handler. To get a better understanding of what is being done run View(imabc:::run_handler) in the console to see how the backend_fun is being used.

Output Files::

If an output directory is specified files are saved for each of the objects returned by the function. They are named as follows:

- Good_SimulatedParameters_tag.csv = good_parm_draws
- Good_SimulatedTargets_tag.csv = good_sim_target
- Good_SimulatedDistances_tag.csv = good_target_dist
- MeanCovariance_tag.csv = mean_cov
- CurrentPriors_tag.csv = priors
- CurrentTargets_tag.csv = targets
- RunMetadata_tag.csv = metaddata

if validate_run = TRUE, includes:

- SimulatedParameters_tag.csv = all_iter_parm_draws
- SimulatedTargets_tag.csv = all_iter_sim_target
- SimulatedDistances_tags.csv = all_iter_target_dist

Value

A list with:

- good_parm_draws - a data.table of valid parameters for the current target bounds
- good_sim_target - a data.table of simulated target results from good_parm_draws parameters
- good_target_dist - a data.table of distances based on simulated good target results
- mean_cov - a data.frame of the means and covariances of parameters for iterations that had more good parameters than N_cov_points
- priors - The prior object with empirical standard deviation from first N_start generated values
- targets - The target object with updated bounds based on calibration
- metaddata - Important info regarding the function inputs and current set of results including current_iteration (the last iteration that completed) and last_draw (the total number of draws simulated during execution)

if validate_run = TRUE, includes:

- all_iter_parm_draws - all parameters generated by the algorithm, even ones that results in target values outside of the current target bounds
- all_iter_sim_target - all simulated target values from the parameters in all_iter_parm_draws
- all_iter_target_dist - all distances based on simulated target results

---

`PriorsSpecification` *Specify the Prior Distributions for All Parameters*

Description

Helper functions that can be used to create an imabc priors object used by imabc().

Usage

```
add_prior(
  ...,
  dist_base_name = NULL,
  density_fn = NULL,
  quantile_fn = NULL,
  parameter_name = NULL
)

define_priors(..., prior_df = NULL)

as.priors(df, ...)
```

Arguments

`...`	Optional. In add_prior: Named inputs to be passed to the RNG functions. In define_priors: The results of add_prior calls - one for each parameter that is being calibrated.
`dist_base_name`	Optional character(1). The base name of the RNG function set (or the column with the dist_base_name info in as.priors) for the prior distribution.
`density_fn`	Optional character(1). The name of the RNG density function (or the column with the density_fn info in as.priors) for the prior distribution.
`quantile_fn`	Optional character(1). The name of the RNG quantile function (or the column with the quantile_fn info in as.priors) for the prior distribution.
`parameter_name`	Optional character(1). The name of the parameter (or the column with the parameter_name info in as.priors).
`prior_df`	Optional data.frame. Priors stored as a data.frame or from the results object of a previous run.
`df`	data.frame. Each parameter should be a row and each column is an input into add_prior. If a given column doesn't relate to a parameter, set its value to NA.

Value

A priors imabc object.

Distribution Specifications

If the user does not provide any RNG functions specifications, they must provide a single value in order to create a fixed parameter. This is not the most efficient method for using a fixed parameter in a model.

If the user only provides one of the RNG functions specifications, these functions will search for the most logical names for the other functions. I.e. if dist_base_name is provided (e.g. unif), these will assume that the user wishes to use paste0("d", dist_base_name) for the density function and paste0("q", dist_base_name) for the quantile function. These functions will make the corresponding guesses if the user provides density_fn or quantile_fn. If density_fn or quantile_fn are provided, they will assume those functions are preferred over any calculated function names.

RNG Input Specifications

These functions will attempt to pass any extra arguments to the RNG functions. These arguments must be named to match the expected inputs not to create errors. If a value's name cannot be matched to an RNG function input, it will be ignored.

min/max are important values to imabc and will always be defined for each parameter. They are used to evaluate whether any simulated parameters are valid. The user can specify values for them if they want. If the user does not specify them they will look at the RNG function and if the RNG has default values for min/max it will use them, otherwise it will use -Inf/Inf respectively. **Warning**: This behavior depends on the RNG functions using min and max as the input names for the min and max values. If the RNG functions use an alternate name for these concepts they will treat them as separate values. An example of this can be found in the truncnorm package which uses a and b for the min and max respectively. For those functions the user would need to specify inputs for a, b, min, and max in order to get a consistent result.

Parameter Names

The user can specify names by either specifying the input parameter_name in add_prior or by setting the result of an add_prior call to a object in define_priors (e.g. define_priors(x1 = add_prior(...))). If the user specifies the parameter_name input and sets add_prior to an object, the parameter_name value will be used. If no name is specified a unique name will be generated automatically.

Examples

```
add_prior(dist_base_name = "norm")
add_prior(density_fn = "dnorm", mean = 50, sd = 10)
add_prior(quantile_fn = "qnorm", min = 0, max = 1)

# x1, x2, and x3 reflect three parameters in the mdoel.
x1 <- add_prior(dist_base_name = "norm")
define_priors(
  x1 = x1,
  x2 = add_prior(density_fn = "dnorm", mean = 50, sd = 10),
  add_prior(parameter_name = "x3", quantile_fn = "qnorm", min = 0, max = 1)
```

```
)

x1_min <- 0.1
x2_min <- 0.5
x1_max <- 0.9
x2_max <- 1.1
df <- data.frame(
  name_var = c("x1", "x2", "x3"),
  dist_var = c("unif", NA, NA),
  density_var = c(NA, "dtruncnorm", NA),
  quantile_var = c(NA, NA, "qnorm"),
  mean = c(NA, 0.75, 0.5),
  sd = c(NA, 0.05, NA),
  min = c(x1_min, x2_min, NA),
  max = c(x1_max, x2_max, NA),
  a = c(NA, x2_min, NA),
  b = c(NA, x2_max, NA)
)
as.priors(
  df,
  parameter_name = "name_var", dist_base_name = "dist_var",
  density_fn = "density_var", quantile_fn = "quantile_var"
)
```

`read_previous_results` *Read Previous Results*

Description

Searches the files found in path for the files saved by an imabc run and reads them into the current environment.

Usage

```
read_previous_results(path, tag = NULL)
```

Arguments

`path`	character(1). The location of files saved during a previous run.
`tag`	Optional character(1). If multiple runs have been saved to a single path, provide the tag that differentiates them.

Value

A list with a priors object, a targets object, and a list of data.frames needed to continue a calibration with imabc().

Note

tag is required if multiple sets of results are stored in a single location.

While the output of this function are necessary for a restart, the user does not need to use this function for restarting a calibration. imabc() handles this function for the user via the previous_results_* input options.

`TargetsSpecifications` *Specify Targets*

Description

Helper functions that can be used to create an imabc targets object used by imabc().

Usage

```
add_target(
  target,
  starting_range,
  stopping_range,
  target_name = NULL,
  FUN = NULL
)

group_targets(..., group_name = NULL)

define_targets(..., target_df = NULL)

as.targets(df, ...)
```

Arguments

`target`	numeric(1). The value a target function is aiming for.
`starting_range`	numeric(2). The initial range of values imabc will consider as good when testing simulated parameters.
`stopping_range`	numeric(2). The range of values a target function's simulated value must be within to be considered calibrated.
`target_name`	Optional character(1). The name of the target.
`FUN`	Optional function. The function that takes parameters and calculated the target value. See Target Function.
`...`	In group_targets: The results of add_target calls - one for each target within a grouping of targets. See Target Groups. In define_targets: The results of add_target and/or group_target calls - one for each target or grouping of targets. In as.targets: alternate column names for the target settings can be any one of target_names, targets, current_lower_bounds, current_upper_bounds, stopping_lower_bounds, or stopping_upper_bounds

group_name	Optional character(1). The name for the grouping of targets.
target_df	Optional data.frame. Targets stored as a data.frame or from the results object of a previous run.
df	data.frame. Each row is a target and the columns represent the different pieces of information relevant to the targets.

Value

A targets imabc object.

Target Values

When specifying values the following condition must always hold true:

```
starting_range[1] <= stopping_range[1] <= target <= stopping_range[2] <= starting_range[2]
```

As imabc simulates parameters, it will test them using the target function(s) against the starting range. Parameters whose values fall within the starting range will be kept through to the next iteration and will be used to generate new parameters for testing. As the parameters get better at falling withing the initial range, imabc will reduce the valid range of targets to be considered. Once the current valid range matches the stopping range the algorithm will no longer reduce the valid range of target values.

Target Groups

A grouped target refers to a set of scalar targets that were gathered as part of the same study or otherwise refer to a series of outcomes, such as outcomes reported by age, by sex, or over time (a time series). When targets are grouped imabc will constrict the range of valid target values relative to the least improved target within the group of targets. This lets the range of simulated parameters stay wide enough to continue improving all the targets.

Target Names

The user can specify names by either specifying the input target_name in add_target or by setting the result of an add_target call to a object in group_targets or define_targets (e.g. group_targets(t1 = add_target(...))). If the user specifies the target_name input and sets add_target to an object, the target_name value will be used. If no name is specified a unique name will be generated automatically.

These same rules also applies to groups of targets and the group_name input in group_targets. However, group_targets can only be added as an input to define_targets. If a single target is added in define_targets it will not have a group name.

Target Function

There are multiple ways to specify a target function. One way is to attach it to the target object using the FUN input in add_target. The inputs to the target function can either be a single object (e.g. function(x)) or several objects whose name is equal to the parameter they represent (e.g. function(x1, x2)). If a single object is used, the user can assume that a name vector with all parameters specified in the priors object will be passed to the function and the order of the vector will be the

same as the order in which they were specified with define_priors. For example, if someone specified three parameters named x1, x3, and x2 respectively then the following specifications would all be equivalent:

```
function(x1, x3) { x1 + x3 } == function(x) { x["x1"] + x["x3"] } == function(x) { x[1] + x[2] }
```

Additionally, for more complex situations the user may also reference the targets object and priors object within a target function but they must specify them as inputs (e.g. function(x, targets, priors)) and use the objects as they exist within those objects. See define_target_function for more details and other ways to specify the target function.

Examples

```
add_target(target = 0.5, starting_range = c(0.2, 0.9), stopping_range = c(0.48, 0.51))
add_target(
  target = 1.5, starting_range = c(1.0, 2.0), stopping_range = c(1.49, 1.51),
  FUN = function(x1, x2) { x1 + x2 + rnorm(1, 0, 0.01) }
)

group_targets(
 targ1 = add_target(target = 0.5, starting_range = c(0.2, 0.9), stopping_range = c(0.48, 0.51)),
  add_target(
    target_name = "targ2",
    target = 1.5, starting_range = c(1.0, 2.0), stopping_range = c(1.49, 1.51),
    FUN = function(x1, x2) { x1 + x2 + rnorm(1, 0, 0.01) }
  )
)
define_targets(
  group1 = group_targets(
   targ1 = add_target(target = 0.5, starting_range = c(0.2, 0.9), stopping_range = c(0.48, 0.51)),
    add_target(
      target_name = "targ2",
      target = 1.5, starting_range = c(1.0, 2.0), stopping_range = c(1.49, 1.51)
    )
  ),
 targ3 = add_target(target = 1, starting_range = c(0.2, 1.9), stopping_range = c(0.9, 1.1))
)

df <- data.frame(
  target_groups = c("G1", "G1", NA),
  target_names = c("T1", "T3", "T2"),
  targets = c(1.5, 0.5, -1.5),
  current_lower_bounds = c(1, 0.2, -2),
  current_upper_bounds = c(2, 0.9, -1),
  stopping_lower_bounds = c(1.49, 0.49, -1.51),
  stopping_upper_bounds = c(1.51, 0.51, -1.49)
)
as.targets(df)
```

Index