Package 'multiRL'

January 26, 2026

Version 0.2.3

Title Reinforcement Learning Tools for Multi-Armed Bandit

Description A flexible general-purpose toolbox for implementing Rescorla-Wagner models in multi-armed bandit tasks.
As the successor and functional extension of the 'binaryRL' package, 'multiRL' modularizes the Markov Decision Process (MDP) into six core components. This framework enables users to construct custom models via intuitive if-else syntax and define latent learning rules for agents.
For parameter estimation, it provides both likelihood-based inference (MLE and MAP) and simulation-based inference (ABC and RNN), with full support for parallel processing across subjects.
The workflow is highly standardized, featuring four main functions that strictly follow the four-step protocol (and ten rules) proposed by Wilson & Collins (2019) <doi:10.7554/eLife.49547>.
Beyond the three built-in models (TD, RSTD, and Utility), users can easily derive new variants by declaring which variables are treated as free parameters.

Maintainer YuKi <hmz1969a@gmail.com>

URL https://yuki-961004.github.io/multiRL/

BugReports https://github.com/yuki-961004/multiRL/issues

License GPL-3

Encoding UTF-8

LazyData TRUE

ByteCompile TRUE

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports methods, utils, Rcpp, compiler, future, doFuture, foreach, doRNG, progressr, ggplot2, scales, grDevices

LinkingTo Rcpp

Suggests stats, GenSA, GA, DEoptim, pso, mlrMBO, mlr, ParamHelpers, smoof, lhs, DiceKriging, rgenoud, cmaes, nloptr, abc, tensorflow, keras, reticulate

NeedsCompilation yes

Author YuKi [aut, cre] (ORCID: <https://orcid.org/0009-0000-1378-1318>),
Xinyu [aut] (ORCID: <https://orcid.org/0009-0004-4974-9191>)

Repository CRAN

Date/Publication 2026-01-26 16:20:14 UTC

Contents

algorithm . . . 3
behrule . . . 4
colnames . . . 5
control . . . 6
data . . . 9
engine_ABC . . . 10
engine_RNN . . . 11
estimate . . . 12
estimate_0_ENV . . . 15
estimate_1_LBI . . . 16
estimate_1_MAP . . . 16
estimate_1_MLE . . . 18
estimate_2_ABC . . . 19
estimate_2_RNN . . . 20
estimate_2_SBI . . . 21
estimation_methods . . . 21
fit_p . . . 23
funcs . . . 24
func_alpha . . . 28
func_beta . . . 30
func_delta . . . 32
func_epsilon . . . 33
func_gamma . . . 35
func_zeta . . . 37
MAB . . . 38
params . . . 39
plot.multiRL.replay . . . 43
policy . . . 44
priors . . . 45
process_1_input . . . 46
process_2_behrule . . . 48
process_3_record . . . 48
process_4_output_cpp . . . 49
process_4_output_r . . . 50
process_5_metric . . . 50
rcv_d . . . 51
rpl_e . . . 53
RSTD . . . 55
run_m . . . 56

settings 58
summary,multiRL.model-method 59
system 60
TAB 61
TD 62
Utility 63

Index **65**

---

algorithm *Algorithm Packages*

Description

The package supports the following eight optimization packages for finding the optimal values of the model's free parameters. Note that if you use "NLOPT", you must consult its official documentation to input a specific algorithm name. If no local search algorithm is specified, the default local search method used will be "NLOPT_LN_BOBYQA".

Class

`algorithm [Character]`

Packages

1. L-BFGS-B (from `stats::optim`)
2. Simulated Annealing (`GenSA::GenSA`)
3. Genetic Algorithm (`GA::ga`)
4. Differential Evolution (`DEoptim::DEoptim`)
5. Bayesian Optimization (`mlrMBO::mbo`)
6. Particle Swarm Optimization (`pso::psoptim`)
7. Covariance Matrix Adapting Evolutionary Strategy (`cmaes::cma_es`)
8. Nonlinear Optimization (`nloptr::nloptr`)

Example

```
# supported algorithms
algorithm = "L-BFGS-B"
algorithm = "GenSA"
algorithm = "GA"
algorithm = "DEoptim"
algorithm = "Bayesian"
algorithm = "PSO"
algorithm = "CMA-ES"
algorithm = "NLOPT_GN_MLSL"
```

behrule *Behavior Rules*

Description

In most instances of the Multi-Armed Bandit (MAB) task, the cue aligns with the response. For example, you are required to select one of four bandits (A, B, C, or D), receive immediate feedback, and subsequently update the expected value of the selected bandit.

When the cue and the response are inconsistent, the agent needs to form a latent rule. For example, in the arrow paradigm of Rmus et al. (2024) doi:10.1371/journal.pcbi.1012119, participants can only choose left or right, but what they actually need to learn is the value associated with arrows of different colors.

The final case represents my personal interpretation, when participants have limited working-memory capacity and an object can be decomposed into many elements, they may update the values of only a subset of those elements rather than the entire object.

Class

`behrule [List]`

Slots

- `cue [CharacterVector]`

  A cue refers to the stimulus—or a component of the stimulus—presented in the paradigm. It represents the internal target the agent selects, which may differ from the actual behavioral response. For instance, cue is the color of arrows, rather than the direction.

- `rsp [CharacterVector]`

  The `rsp` represents the action the agent actually makes. It typically has a mapping relationship with the cue. For example, in the arrow paradigm of Rmus et al. (2024) doi:10.1371/journal.pcbi.1012119, the agent updates the value associated with the arrow's color, but the overt response is the direction corresponding to the currently chosen color arrow.

Example

```
# latent rule
behrule = list(
  cue = c("Red", "Yellow", "Green", "Blue"),
  rsp = c("Up", "Down", "Left", "Right")
)
```

References

Rmus, M., Pan, T. F., Xia, L., & Collins, A. G. (2024). Artificial neural networks for model identification and parameter estimation in computational cognitive models. *PLOS Computational Biology, 20*(5), e1012119. doi:10.1371/journal.pcbi.1012119

`colnames` *Column Names*

Description

Users must categorize and inform the program of the column names within their dataset.

Class

`colnames [List]`

Slots

1. `subid [Character]`
   The column name of subject identifier.
   Column name that is exactly "Subject" can be recognized automatically.
2. `block[Character]`
   The column name of block index.
   Column name that is exactly "Block" can be recognized automatically.
3. `trial[Character]`
   The column name of trial index.
   Column name that is exactly "Trial" can be recognized automatically.
4. `object [CharacterVector]`
   The column names of objects presented in the task, with individual elements separated by underscores ("_").
   Column names that are prefixed with "Object_" can be recognized automatically.
5. `reward [CharacterVector]`
   The column names of the reward associated with each object; ensure that every object has its own corresponding reward.
   Column names that are prefixed with "Reward_" can be recognized automatically.
6. `action [Character]`
   The column name of the action taken by the agent, which must match an object or one of its elements.
   Column name that is exactly "Action" can be recognized automatically.
7. `exinfo [CharacterVector]`
   The column names of extra information that the model may use during the markov decision process.

Tips

Users can use these variables within the model's functions. see tutorial.

Example

```
# column names
colnames = list(
  subid = "Subject",
  block = "Block",
  trial = "Trial",
  object = c("Object_1", "Object_2", "Object_3", "Object_4"),
  reward = c("Reward_1", "Reward_2", "Reward_3", "Reward_4"),
  action = "Action",
  exinfo = c("Frame", "NetWorth", "RT", "Mood")
)
```

`control` *Control Algorithm Behavior*

Description

The `control` argument is a mandatory list used to customize and manage various aspects of the iterative process, covering everything from optimization settings to model configuration.

Class

`control [List]`

Note

Different estimation methods require different slots. However, there is no need to worry if you set unnecessary slots, as this will not affect the execution.

1. Likelihood Based Inference (LBI)

- `sample [int]`

  This parameter denotes the quantity of simulated data generated during the parameter recovery process.
- `iter [int]`

  This parameter defines the maximum number of iterations. The iterative process will stop when this value is reached. The default value is `10`. It is recommended that you set this value to at least `100` for formal fitting procedures.
- `pars [NumericVector]`

  Some algorithms require the specification of initial iteration values. If this value is left as the default NA, the iteration will commence with an initial value set to the lower bound of the estimate plus `0.01`.

- `dash` `[Numeric]`

  To prevent the optimal parameter estimates from converging to boundary values when the number of iterations is insufficient, a small value is added to the lower bound and subtracted from the upper bound.

  For instance, if the input parameter bounds are `(0, 1)`, the actual bounds used for fitting will be `[0.00001, 0.99999]`. This design prevents the occurrence of Infinite values.

- `size` `[int]`

  Some algorithms, such as Genetic Algorithms (`GA`), require the specification of initial population values. For the definition of the population, users may refer to the relevant documentation on evolutionary algorithms. The default value is consistent with the standard default in `GA`, which is `50`.

- `seed` `[int]`

  The random seed controls the reproducibility of each iteration. Specifically, it determines how the algorithm package generates "random" input parameters when searching for the optimal parameters. Fixing the seed ensures that the optimal parameters found are the same in every run. The default value is `123`.

- `core` `[int]`

  Since the parameter fitting process for individual subjects is independent, this procedure can be accelerated using CPU parallelism. This argument specifies the number of subjects to be fitted simultaneously (the number of parallel threads), with the default set to `1`. If the user wishes to speed up the fitting, they can increase the number of cores appropriately based on their system specifications.

1.1 Maximum Likelihood Estimation (MLE):

- Nothing special

1.2 Maximum A Posteriori (MAP):

- `diff` `[double]`
  In the Expectation–Maximization with Maximum A Posteriori algorithm (EM-MAP), after estimating the optimal parameters for all subjects in each iteration, the posterior distribution of each free parameter is calculated, followed by continuous refinement of the prior distribution. The process stops when the change in the log-posterior value is less than the `diff`, which defaults to `0.001`.
- `patience` `[int]`
  Given that the Expectation–Maximization with Maximum A Posteriori (EM-MAP) process can be time-consuming and often encounters non-convergence issues—for instance, when the log-posterior oscillates around a certain value—the patience parameter is used to manage early termination.Specifically, patience is incremented by 1 when the current result is better than the best previous result, and decremented by 1 when it is worse. The iteration is prematurely terminated when the patience count reaches zero.

2. Simulation Based Inference (SBI)

- `sample` `[int]`

  This parameter denotes the quantity of simulated data generated during the parameter recovery process.

- `train [int]`

  This parameter is used to specify the quantity of simulated data utilized when training the Approximate Bayesian Computation (ABC) or Recurrent Neural Network (RNN) models.

- `scope [Character]`

  This parameter can be defined as `individual` or `shared`. The former indicates that a separate Approximate Bayesian Computation (ABC) or Recurrent Neural Network (RNN) model is trained for each dataset, while the latter means that only one Approximate Bayesian Computation (ABC) or Recurrent Neural Network (RNN) model is trained and shared across all datasets. In the context of the `rcv_d` function, the default setting is `"shared"`, whereas in `fit_p`, the default is `"individual"`.

- `seed [int]`

  When performing parameter recovery using Simulation-Based Inference (SBI) estimation methods, two sets of simulated data are involved: one used to generate the data for recovery, and another used to train the Approximate Bayesian Computation (ABC) or Recurrent Neural Network (RNN) models. To guarantee the independence of these two datasets, the seed for generating the training data is automatically multiplied by 2.

- `core [int]`

  Since the parameter fitting process for individual subjects is independent, this procedure can be accelerated using CPU parallelism. This argument specifies the number of subjects to be fitted simultaneously (the number of parallel threads), with the default set to `1`. If the user wishes to speed up the fitting, they can increase the number of cores appropriately based on their system specifications. When `estimate = "RNN"`, since model training is typically handled by the GPU, setting `core > 1` will only accelerate the generation of simulated data.

2.1 Approximate Bayesian Computation (ABC):

- `tol [double]`

  This parameter, aka tolerance, controls how strict the Approximate Bayesian Computation (ABC) algorithm is when selecting good simulated data. It sets the acceptance rate. For example, setting `tol = 0.1` (the default) means only the 10 percent of simulated data that is closest to your actual data is used.

2.2 Recurrent Neural Network (RNN):

- `info [CharacterVector]`

  The Recurrent Neural Network (RNN) needs to find the mapping relationship between the dataset and the free parameters. To minimize the time required for this process, we should only include useful information in the input dataset. The `info` parameter accepts a character vector which represents the amount of information (i.e., the specific columns) you deem necessary for training the Recurrent Neural Network (RNN) model. By default, only the `colnames$object` and `colnames$action` columns are included as input.

- `layer [Character]`

  Recurrent Neural Networks (RNNs) are neural networks where the sequence order is meaningful. Currently, the package supports two types of recurrent layers: Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM). You can specify either of these as the recurrent layer in your model.

- `units [int]`

  The number of neurons (or units) in the Recurrent Layer (GRU or LSTM). Conceptually, this parameter represents the memory capacity and complexity of the network; it dictates how much information about the sequential trials the model can store and process.

- `batch_size [int]`
  The number of samples processed before the model's parameters are updated. Think of this as the size of a study group; the model reviews this batch of data before adjusting its internal weights. A larger batch size speeds up calculation but may lead to less optimal convergence.
- `epochs [int]`
  The number of times the learning algorithm will work through the entire training dataset. This is equivalent to running through the "textbook" multiple times. Each epoch means the model has seen every training sample once. More epochs allow for more training but increase the risk of overfitting.

Example

```
# default values
control = list(
  # LBI
  pars = NA,
  dash = 1e-5,
  iter = 10,
  size = 50,
  seed = 123,
  core = 1,
  # MLE
  ...,
  # MAP
  diff = 0.001,
  patience = 10,
  # SBI
  sample = 100,
  train = 1000,
  scope = "individual",
  # ABC
  tol = 0.1,
  #
  info = c(colnames$object, colnames$action),
  layer = "GRU",
  units = 128,
  batch_size = 10,
  epochs = 100
)
```

`data` *Dataset Structure*

Description

Experimental data from any Multi-Armed Bandit (MAB)-like task.

Class

`data [data.frame]`

subid	block	trial	object_1	object_2	object_3	object_4	reward_1	reward_2	reward_3	reward_4	action
1	1	1	A	B	C	D	20	0	60	40	A
1	1	2	A	B	C	D	20	40	60	80	B
1	1	3	A	B	C	D	20	0	60	40	C
1	1	4	A	B	C	D	20	40	60	80	D
..	..	..	..	..	..	..	..	..	..	..	..

Details

Each row must contain all information relevant to that trial for running a decision-making task (e.g., multi-armed bandit) as well as the feedback received.

In this type of paradigm, the rewards associated with possible actions must be explicitly written in the table for every trial (aka, tabular case, see Sutton & Barto, 2018, Chapter 2).

Note

The package does not perform any real-time random sampling based on the agent's choices; therefore, Users should pre-define the reward for each possible action in every trial.

You should never ever ever use true randomization to generate rewards.

Doing so would result in different participants interacting with multi-armed bandits that do not share the same expected values. In such cases, if two participants show different parameter estimates in a same model, we cannot determine whether the difference reflects stable individual traits or simply the fact that one participant happened to be lucky while the other was not.

References

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed). MIT press.

`engine_ABC` *The Engine of Approximate Bayesian Computation (ABC)*

Description

Because `abc::abc()` requires summary statistics together with the corresponding input parameters, this function converts the generated simulated data into a standardized collection of summary statistics and input parameters, facilitating subsequent execution of `abc::abc()`.

Usage

```
engine_ABC(
  data,
  colnames,
  behrule,
  model,
  funcs = NULL,
  priors,
  settings = NULL,
  control = control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`model`	Reinforcement Learning Model
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

A `List` containing a `DataFrame` of the parameters used to generate the simulated data and the summary statistics for Approximate Bayesian Computation (ABC).

`engine_RNN` *The Engine of Recurrent Neural Network (RNN)*

Description

Because TensorFlow requires numeric arrays and input parameters to learn the mapping between them when building a Recurrent Neural Network (RNN) model, this function transforms simulated data into a standardized dataset and invokes TensorFlow to train the model.

Usage

```
engine_RNN(
  data,
  colnames,
  behrule,
  model,
  funcs = NULL,
  priors,
  settings = NULL,
  control = control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`model`	Reinforcement Learning Model
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

A specialized TensorFlow-trained Recurrent Neural Network (RNN) object. The model can be used with the `predict()` function to make predictions on a new data frame, estimating the input parameters that are most likely to have generated the given dataset.

`estimate` *Estimate Methods*

Description

The method used for parameter estimation, including `"MLE"` (Maximum Likelihood Estimation), `"MAP"` (Maximum A Posteriori), `"ABC"` (Approximate Bayesian Computation), and `"RNN"` (Recurrent Neural Network).

Class

`estimate [Character]`

1. Likelihood Based Inference (LBI)

This estimation approach is adopted when latent rules are absent and human behavior aligns with the value update objective. In other words, it is the estimation method employed when the log-likelihood can be calculated.

1.1 Maximum Likelihood Estimation (MLE): Log-likelihood reflects the similarity between the human's observed choice and the model's prediction. The free parameters (e.g., learning rate) govern the entire Markov Decision Process, thereby controlling the returning log-likelihood value. Maximum Likelihood Estimation (MLE) then involves finding the set of free parameters that maximizes the sum of the log-likelihoods across all trials.

The search for these optimal parameters can be accomplished using various algorithms. For details, please refer to the documentation for algorithm.

1. The Markov Decision Process (MDP) continuously updates the expected value of each action.
2. These expected values are transformed into action probabilities using the soft-max function.
3. The log-probability of each action is calculated.
4. The likelihood is defined as the product of the human actions and the log-probabilities estimated by the model.

1.2 Maximum A Posteriori (MAP):

Maximum A Posteriori (MAP) is an extension of Maximum Likelihood Estimation (MLE) In addition to optimizing parameters for each individual subject based on the likelihood, Maximum A Posteriori incorporates information about the population distribution of the parameters.

1. Perform an initial Maximum Likelihood Estimation (MLE) to find the best-fitting parameters for each individual subject.
2. Use these best-fitting parameters to estimate the Probability Density Function of the population-level parameter distribution. (The Expectation–Maximization with Maximum A Posteriori estimation (EM-MAP) framework is inspired by the `sjgershm/mfit`. However, unlike `mfit`, which typically assumes a normal distribution for the posterior. In my opinion, the posterior density is derived based on the specific prior distribution. For example, if the prior follows an exponential distribution, the estimation remains within the exponential family rather than being forced into a normal distribution.)
3. Perform Maximum Likelihood Estimation (MLE) again for each subject. However, instead of returning the log-likelihood, the returned value is the log-posterior. In other words, this step considers the probability of the best-fitting parameter occurring within its derived population distribution. This penalization helps avoid finding extreme parameter estimates.
4. The above steps are repeated until the log-posterior converges.

2. Simulation Based Inference (SBI)

Simulation-Based Inference (SBI) can be employed when calculating the log-likelihood is impossible or computationally intractable. Simulation-Based Inference (SBI) generally seeks to establish a direct relationship between the behavioral data and the parameters, without compressing the behavioral data into a single value (log-likelihood).

2.1 Approximate Bayesian Computation (ABC):

The Approximate Bayesian Computation (ABC) model is trained by finding a mapping between the summary statistics and the free parameters. Once the model is trained, given a new set of summary statistics, the model can instantly determine the corresponding input parameters.

1. Generate a large amount of simulated data using randomly sampled input parameters.
2. Compress the simulated data into summary statistics—for instance, by calculating the proportion of times each action was executed within different blocks.
3. Establish the mapping between these summary statistics and the input parameters, which constitutes training the Approximate Bayesian Computation (ABC) model.
4. Given a new set of summary statistics, the trained model outputs the input parameters most likely to have generated those statistics.

2.2 Recurrent Neural Network (RNN):

The Recurrent Neural Network (RNN) directly seeks a mapping between the simulated dataset itself and the input free parameters. When provided with new behavioral data, the trained model can estimate the input parameters most likely to have generated that specific dataset.

- The Recurrent Neural Network (RNN) component included in `multiRL` is merely a shell for TensorFlow. Consequently, users who intend to use `estimate = "RNN"` must first install TensorFlow.

The Recurrent Neural Network (RNN) model is trained using only `state` and `action` data as the raw dataset by default. In other words, the developer assumes that the only necessary input information for the Recurrent Neural Network (RNN) comprises the trial-by-trial object presentation (the state) and the agent's resultant action. This constraint is adopted because excessive input information may not only interfere with model training but also lead to unnecessary time consumption.

1. The raw simulated data is limited to the state (object information presented on each trial) and the action chosen by the agent in response to that state.
2. After the simulated data is generated, it is partitioned into a training set and a validation set, and the RNN training commences.
3. The iteration stops when both the training and validation sets converge. If the Mean Squared Error (MSE) of the validation set is high while the MSE of the training set is low, this indicates overfitting, suggesting that the Recurrent Neural Network (RNN) model may lack generalization ability.
4. Given a new dataset, the trained model infers the input parameters that are most likely to have generated that dataset.

Example

```
# supported estimate methods
# Maximum Likelihood Estimation
estimate = "MLE"
# Maximum A Posteriori
estimate = "MAP"
# Approximate Bayesian Computation
estimate = "ABC"
# Recurrent Neural Network
estimate = "RNN"
```

estimate_0_ENV *Tool for Generating an Environment for Models*

Description

This function creates an independent R environment for each model (or object function) when searching for optimal parameters using an algorithm package. Such isolation is especially important when parameter optimization is performed in parallel across multiple subjects. The function transfers standardized input parameters into a dedicated environment, ensuring that each model is evaluated in a self-contained and interference-free context.

Usage

```
estimate_0_ENV(
  data,
  colnames = list(),
  behrule,
  funcs = list(),
  priors = list(),
  settings = list(),
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`...`	Additional arguments passed to internal functions.

Value

An `environment`, `multiRL.env` contains all variables required by the objective function and is used to isolate environments during parallel computation.

estimate_1_LBI *Likelihood-Based Inference (LBI)*

Description

This function provides a unified interface to multiple algorithm packages, allowing different optimization algorithms to be selected for estimating optimal model parameters. The entire optimization framework is based on the log-likelihood returned by the model (or object function), making this function a collection of likelihood-based inference (LBI) methods. By abstracting over algorithm-specific implementations, the function enables flexible and consistent parameter estimation across different optimization backends.

Usage

```
estimate_1_LBI(model, env, algorithm, lower, upper, control = list(), ...)
```

Arguments

`model`	Reinforcement Learning Model
`env`	multiRL.env
`algorithm`	Algorithm packages that multiRL supports, see algorithm
`lower`	Lower bound of free parameters
`upper`	Upper bound of free parameters
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.model` generated using the estimated optimal parameters.

estimate_1_MAP *Estimation Method: Maximum A Posteriori (MAP)*

Description

This function first performs a maximum likelihood estimation (MLE) to obtain the best-fitting parameters for all subjects based on maximum likelihood. It then computes the likelihood-based posterior using user-specified prior distributions. Based on the current group-level data, the prior distributions are subsequently updated. This procedure is iteratively repeated until the likelihood-based posterior converges. The entire process is referred to as Expectation–Maximization with Maximum A Posteriori estimation(EM-MAP).

Usage

```
estimate_1_MAP(
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors,
  settings = NULL,
  algorithm,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`algorithm`	Algorithm packages that multiRL supports, see algorithm
`lowers`	Lower bound of free parameters in each model.
`uppers`	Upper bound of free parameters in each model.
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `DataFrame` containing, for each model, the estimated optimal parameters and associated model fit metrics.

estimate_1_MLE *Estimation Method: Maximum Likelihood Estimation (MLE)*

Description

This function essentially applies `estimate_1_LBI()` to each subject's data, estimating subject-specific optimal parameters based on maximum likelihood. Because the fitting process for each subject is independent, the procedure can be accelerated using parallel computation.

Usage

```
estimate_1_MLE(
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors,
  settings = NULL,
  algorithm,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`algorithm`	Algorithm packages that multiRL supports, see algorithm
`lowers`	Lower bound of free parameters in each model.
`uppers`	Upper bound of free parameters in each model.
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `DataFrame` containing, for each model, the estimated optimal parameters and associated model fit metrics.

`estimate_2_ABC` *Estimation Method: Approximate Bayesian Computation (ABC)*

Description

This function takes a large set of simulated data to train an Approximate Bayesian Computation (ABC) model and then uses the trained model to estimate optimal parameters for the target data.

Usage

```
estimate_2_ABC(
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors,
  settings = NULL,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`lowers`	Lower bound of free parameters in each model.
`uppers`	Upper bound of free parameters in each model.
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `DataFrame` containing, for each model, the estimated optimal parameters and associated model fit metrics.

`estimate_2_RNN` *Estimation Method: Recurrent Neural Network (RNN)*

Description

This function takes a large set of simulated data to train an Recurrent Neural Network (RNN) model and then uses the trained model to estimate optimal parameters for the target data.

Usage

```
estimate_2_RNN(
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors,
  settings,
  control,
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `DataFrame` containing, for each model, the estimated optimal parameters and associated model fit metrics.

estimate_2_SBI *Simulated-Based Inference (SBI)*

Description

Since both Approximate Bayesian Computation (ABC) and Recurrent Neural Network (RNN) are simulation-based inference methods, they require a model built from a large amount of simulated data before running. This model is then used to predict the most likely input parameters corresponding to the real data. Therefore, this function generates random input parameters using user-specified distributions and produces simulated data based on these parameters.

Usage

```
estimate_2_SBI(model, env, priors, control = list(), ...)
```

Arguments

`model`	Reinforcement Learning Model
`env`	multiRL.env
`priors`	Prior probability density function of the free parameters, see priors
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

A `List` containing, for each model, simulated data generated using randomly sampled parameters.

estimation_methods *Estimate Methods*

Description

This function provides a unified interface for four estimation methods: Maximum Likelihood Estimation (MLE), Maximum A Posteriori (MAP), Approximate Bayesian Computation (ABC), and Recurrent Neural Network (RNN), allowing users to execute different methods simply by setting `estimate = "???"`.

Usage

```
estimation_methods(
  estimate,
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors = NULL,
  settings = NULL,
  algorithm,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

`estimate`	Estimate method that you want to use, see estimate
`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`algorithm`	Algorithm packages that multiRL supports, see algorithm
`lowers`	Lower bound of free parameters in each model.
`uppers`	Upper bound of free parameters in each model.
`control`	Settings manage various aspects of the iterative process, see control
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `DataFrame` containing, for each model, the estimated optimal parameters and associated model fit metrics.

fit_p *Step 3: Optimizing parameters to fit real data*

Description

Step 3: Optimizing parameters to fit real data

Usage

```
fit_p(
  estimate,
  data,
  colnames,
  behrule,
  ids = NULL,
  funcs = NULL,
  priors = NULL,
  settings = NULL,
  models,
  algorithm,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

estimate	Estimate method that you want to use, see estimate
data	A data frame in which each row represents a single trial, see data
colnames	Column names in the data frame, see colnames
behrule	The agent's implicitly formed internal rule, see behrule
ids	The Subject ID of the participant whose data needs to be fitted.
funcs	The functions forming the reinforcement learning model, see funcs
priors	Prior probability density function of the free parameters, see priors
settings	Other model settings, see settings
models	Reinforcement Learning Models
algorithm	Algorithm packages that multiRL supports, see algorithm
lowers	Lower bound of free parameters in each model.
uppers	Upper bound of free parameters in each model.
control	Settings manage various aspects of the iterative process, see control
...	Additional arguments passed to internal functions.

Value

An S3 object of class `multiRL.fitting`. A `List` containing, for each model, the estimated optimal parameters and associated model fit metrics.

Example

```
# fitting
fitting.MLE <- multiRL::fit_p(
  estimate = "MLE",

  data = multiRL::TAB,
  colnames = list(
    object = c("L_choice", "R_choice"),
    reward = c("L_reward", "R_reward"),
    action = "Sub_Choose"
  ),
  behrule = list(
    cue = c("A", "B", "C", "D"),
    rsp = c("A", "B", "C", "D")
  ),

  models = list(multiRL::TD, multiRL::RSTD, multiRL::Utility),
  settings = list(name = c("TD", "RSTD", "Utility")),

  algorithm = "NLOPT_GN_MLSL",
  lowers = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
  uppers = list(c(1, 5), c(1, 1, 5), c(1, 5, 1)),
  control = list(core = 10, iter = 100)
)
```

`funcs` *Core Functions*

Description

The Markov Decision Process (MDP) underlying Reinforcement Learning can be decomposed into six fundamental components. By modifying these six functions, an immense number of distinct Reinforcement Learning models can be created. Users only need to grasp the basic Markov Decision Process process and subsequently tailor these six functions to construct a unique reinforcement learning model.

Class

`funcs [List]`

Details

- Action Select
  - Step 1: Agent uses `bias_func` to apply a bias term to the value of each option.
  - Step 2: Agent uses `expl_func` to decide whether to make a purely random exploratory choice.
  - Step 3: Agent uses `prob_func` to compute the selection probability for each action.
- Value Update
  - Step 4: Agent uses `util_func` to translate the objective reward into subjective utility.
  - Step 5: Agent uses `dcay_func` to regress the values of unchosen options toward a baseline.
  - Step 6: Agent uses `rate_func` to update the value of the chosen option.

Learning Rate (α)

`rate_func` is the function that determines the learning rate (α). This function governs how the model selects the α. For instance, you can set different learning rates for different circumstances. Rather than 'learning' in a general sense, the learning rate determines whether the agent updates its expected values (Q-values) using an aggressive or conservative step size across different conditions.

$$Q_{new} = Q_{old} + \alpha \cdot (R - Q_{old})$$

Soft-Max (β)

`prob_func` is the function defined by the inverse temperature parameter (β) and the lapse parameter.

The inverse temperature parameter governs the randomness of choice. If β approaches 0, the agent will choose between different actions completely at random. As β increases, the choice becomes more dependent on the expected value (Q_t), meaning actions with higher expected values have a proportionally higher probability of being chosen.

Note: This formula includes a normalization of the (Q_t) values.

$$P_t(a) = \frac{\exp\left(\beta \cdot (Q_t(a) - \max_j Q_t(a_j))\right)}{\sum_{i=1}^{k} \exp\left(\beta \cdot (Q_t(a_i) - \max_j Q_t(a_j))\right)}$$

The function below, which incorporates the constant lapse rate, is a correction to the standard soft-max rule. This is designed to prevent the probability of any action from becoming exactly 0 (Wilson and Collins, 2019 doi:10.7554/eLife.49547). When the lapse parameter is set to 0.01, every action has at least a 1% probability of being executed. If the number of available actions becomes excessively large (e.g., greater than 100), it would be more appropriate to set the lapse parameter to a much smaller value.

$$P_t(a) = (1 - lapse \cdot N_{shown}) \cdot P_t(a) + lapse$$

Utility Function (γ)

`util_func` is defined by the utility exponent parameter (γ). Its purpose is to account for the fact that the objective reward received by human may hold a different subjective value (utility) across different subjects.

Note: The built-in function is formulated according to Stevens' power law.

$$U(R) = R^{\gamma}$$

Upper Confidence Bound (δ)

`bias_func` is the function defined by the parameter (δ). This function signifies that the expected value of an action is not solely determined by the received reward, but is also influenced by the number of times the action has been executed. Specifically, an action that has been executed fewer times receives a larger exploration bias. (Sutton and Barto, 2018) This mechanism prompts exploration and ensures the agent to execute every action at least once.

$$\text{Bias} = \delta \cdot \sqrt{\frac{\log(N + e)}{N + 10^{-10}}}$$

Epsilon–First, Greedy, Decreasing (ϵ)

`expl_func` is the function defined by the parameter (ϵ) and the constant threshold. This function controls the probability with which the agent engages in exploration (i.e., making a random choice) versus exploitation (i.e., making a value-based choice).

$\epsilon\breve{}first$: The agent must choose randomly for a fixed number of initial trials. Once the number of trials exceeds the threshold, the agent must exclusively choose based on value.

$$P(x) = \begin{cases} i \leq \text{threshold}, & x = 1 \\ i > \text{threshold}, & x = 0 \end{cases}$$

$\epsilon\breve{}greedy$: The agent performs a random choice with probability ϵ and makes a value-based choice with probability $1 - \epsilon$.

$$P(x) = \begin{cases} \epsilon, & x = 1 \\ 1 - \epsilon, & x = 0 \end{cases}$$

$\epsilon\breve{}decreasing$: The probability of making a random choice gradually decreases as the number of trials increases throughout the experiment.

$$P(x) = \begin{cases} \frac{1}{1+\epsilon \cdot i}, & x = 1 \\ \frac{\epsilon \cdot i}{1+\epsilon \cdot i}, & x = 0 \end{cases}$$

Working Memory (ζ)

dcay_func is the function defined by the decay rate parameter (ζ) and the constant bonus. This function indicates that at the end of each trial, not only the value of the chosen option will be changed according to the learning rate, but also the values of the unchosen options also undergo change.

It is due to the limitations of working memory capacity, the values of the unchosen options are hypothesized to decay back towards their initial value at a rate determined by the decay rate parameter (ζ) (Collins and Frank, 2012 doi:10.1111/j.14609568.2011.07980.x).

$$W_{new} = W_{old} + \zeta \cdot (W_0 - W_{old})$$

Furthermore, Hitchcock, Kim and Frank, (2025) doi:10.1037/xge0001817 suggest that if the feedback of the chosen option provides information relevant to the unchosen options, this decay rate may be enhanced or mitigated by the constant bonus.

Example

```
# inner functions
funcs = list(
  # Learning Rate
  rate_func = multiRL::func_alpha,
  # Inverse Temperature
  prob_func = multiRL::func_beta,
  # Utility Function (Stevens' Power Law)
  util_func = multiRL::func_gamma,
  # Upper-Confidence-Bound
  bias_func = multiRL::func_delta,
  # Epsilon-First, Greedy, Decreasing
  expl_func = multiRL::func_epsilon,
  # Working Memory System
  dcay_func = multiRL::func_zeta
)
```

References

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed). MIT press.

Collins, A. G., & Frank, M. J. (2012). How much of reinforcement learning is working memory, not reinforcement learning? A behavioral, computational, and neurogenetic analysis. *European Journal of Neuroscience, 35*(7), 1024-1035. doi:10.1111/j.14609568.2011.07980.x

Wilson, R. C., & Collins, A. G. (2019). Ten simple rules for the computational modeling of behavioral data. *Elife, 8*, e49547. doi:10.7554/eLife.49547

Hitchcock, P. F., Kim, J., Frank, M. J. (2025). How working memory and reinforcement learning interact when avoiding punishment and pursuing reward concurrently. *Journal of Experimental Psychology: General*. doi:10.1037/xge0001817

func_alpha *Function: Learning Rate*

Description

$$Q_{new} = Q_{old} + \alpha \cdot (R - Q_{old})$$

Usage

```
func_alpha(qvalue, reward, params, system, ...)
```

Arguments

`qvalue`	The expected Q values of different behaviors produced by different systems when updated to this trial.
`reward`	The feedback received by the agent from the environment at trial(t) following the execution of action(a)
`params`	Parameters used by the model's internal functions, see params
`system`	When the agent makes a decision, is a single system at work, or are multiple systems involved? see system
`...`	It currently contains the following information; additional information may be added in future package versions.

- idinfo:
  - subid
  - block
  - trial
- exinfo: contains information whose column names are specified by the user.
  - Frame
  - RT
  - NetWorth
  - ...
- behave: includes the following:
  - action: the behavior performed by the human in the given trial.
  - latent: the object updated by the agent in the given trial.
  - simulation: the actual behavior performed by the agent.

Value

A `NumericVector` containing the updated values computed based on the learning rate.

Body

```
func_alpha <- function(
    qvalue,
    reward,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]

  alpha     <-  params[["alpha"]]
  alphaN    <-  params[["alphaN"]]
  alphaP    <-  params[["alphaP"]]

  # Determine the model currently in use based on which parameters are free.
  if (
    system == "RL" && !(is.null(alpha)) && is.null(alphaN) && is.null(alphaP)
  ) {
    model <- "TD"
  } else if (
    system == "RL" && is.null(alpha) && !(is.null(alphaN)) && !(is.null(alphaP))
  ) {
    model <- "RSTD"
  } else if (
    system == "WM"
  ) {
    model <- "WM"
    alpha <- 1
  } else {
    stop("Unknown Model! Plase modify your learning rate function")
  }

  # TD
  if (model == "TD") {
    update <- qvalue + alpha   * (reward - qvalue)
  # RSTD
  } else if (model == "RSTD" && reward < qvalue) {
    update <- qvalue + alphaN * (reward - qvalue)
  } else if (model == "RSTD" && reward >= qvalue) {
    update <- qvalue + alphaP * (reward - qvalue)
  # WM
```

```
  } else if (model == "WM") {
    update <- qvalue + alpha  * (reward - qvalue)
  }

  return(update)
}
```

---

func_beta *Function: Soft-Max*

Description

$$P_t(a) = \frac{\exp(\beta \cdot (Q_t(a) - \max_{a' \in \mathcal{A}} Q_t(a')))}{\sum_{a' \in \mathcal{A}} \exp(\beta \cdot (Q_t(a') - \max_{a'_i \in \mathcal{A}} Q_t(a'_i)))}$$

$$P_t(a) = (1 - lapse \cdot N_{shown}) \cdot P_t(a) + lapse$$

Usage

```
func_beta(qvalue, explor, params, system, ...)
```

Arguments

qvalue	The expected Q values of different behaviors produced by different systems when updated to this trial.
explor	Whether the agent made a random choice (exploration) in this trial.
params	Parameters used by the model's internal functions, see params
system	When the agent makes a decision, is a single system at work, or are multiple systems involved? see system
...	It currently contains the following information; additional information may be added in future package versions.

- idinfo:
  - subid
  - block
  - trial
- exinfo: contains information whose column names are specified by the user.
  - Frame
  - RT
  - NetWorth
  - ...
- behave: includes the following:
  - action: the behavior performed by the human in the given trial.
  - latent: the object updated by the agent in the given trial.
  - simulation: the actual behavior performed by the agent.

Value

A NumericVector containing the probability of choosing each option.

Body

```
func_beta <- function(
    qvalue,
    explor,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]

  beta     <- params[["beta"]]
  lapse    <- params[["lapse"]]
  weight   <- params[["weight"]]
  capacity <- params[["capacity"]]
  sticky   <- params[["sticky"]]

  index     <- which(!is.na(qvalue[[1]]))
  n_shown   <- length(index)
  n_system  <- length(qvalue)
  n_options <- length(qvalue[[1]])

  # Assign weights to different systems
  if (length(weight) == 1L) {weight <- c(weight, 1 - weight)}
  weight <- weight / sum(weight)
  if (n_system == 1) {weight <- weight[1]}

  # Compute the probabilities estimated by different systems
  prob_mat <- matrix(0, nrow = n_options, ncol = n_system)

  if (explor == 1) {
    prob_mat[index, ] <- 1 / n_shown
  } else {
    for (s in seq_len(n_system)) {
      sub_qvalue <- qvalue[[s]]
      exp_stable <- exp(beta * (sub_qvalue - max(sub_qvalue, na.rm = TRUE)))
      prob_mat[, s] <- exp_stable / sum(exp_stable, na.rm = TRUE)
    }
```

```
  }

  # Weighted average
  prob <- as.vector(prob_mat

  # lapse
  prob <- (1 - lapse * n_shown) * prob + lapse

  return(prob)
}
```

`func_delta` *Function: Upper-Confidence-Bound*

Description

$$\text{Bias} = \delta \cdot \sqrt{\frac{\log(N + e)}{N + 10^{-10}}}$$

Usage

```
func_delta(count, params, ...)
```

Arguments

`count`	How many times this action has been executed
`params`	Parameters used by the model's internal functions, see params
`...`	It currently contains the following information; additional information may be added in future package versions.

- idinfo:
  - subid
  - block
  - trial
- exinfo: contains information whose column names are specified by the user.
  - Frame
  - RT
  - NetWorth
  - ...
- behave: includes the following:
  - action: the behavior performed by the human in the given trial.
  - latent: the object updated by the agent in the given trial.
  - simulation: the actual behavior performed by the agent.

Value

A `NumericVector` containing the bias for each option based on the number of times it has been selected.

Body

```
func_delta <- function(
    count,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]

  delta     <-  params[["delta"]]

  bias <- delta * sqrt(log(count + exp(1)) / (count + 1e-10))

  return(bias)
}
```

`func_epsilon` *Function: ϵ–first, Greedy, Decreasing*

Description

$\epsilon\breve{}first$:

$$P(x) = \begin{cases} i \leq \text{threshold}, & x = 1 \\ i > \text{threshold}, & x = 0 \end{cases}$$

$\epsilon\breve{}greedy$:

$$P(x) = \begin{cases} \epsilon, & x = 1 \\ 1 - \epsilon, & x = 0 \end{cases}$$

$\epsilon\breve{}decreasing$:

$$P(x) = \begin{cases} \frac{1}{1+\epsilon \cdot i}, & x = 1 \\ \frac{\epsilon \cdot i}{1+\epsilon \cdot i}, & x = 0 \end{cases}$$

Usage

```
func_epsilon(rownum, params, ...)
```

Arguments

`rownum`	The trial number
`params`	Parameters used by the model's internal functions, see params
`...`	It currently contains the following information; additional information may be added in future package versions.

- idinfo:
  - subid
  - block
  - trial
- exinfo: contains information whose column names are specified by the user.
  - Frame
  - RT
  - NetWorth
  - ...
- behave: includes the following:
  - action: the behavior performed by the human in the given trial.
  - latent: the object updated by the agent in the given trial.
  - simulation: the actual behavior performed by the agent.

Value

An `int`, either 0 or 1, indicating exploration or exploitation on the current trial.

Body

```
func_epsilon <- function(
    rownum,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]

  epsilon   <-  params[["epsilon"]]
  threshold <-  params[["threshold"]]
```

```
  # Determine the model currently in use based on which parameters are free.
  if (is.na(epsilon) && threshold > 0) {
    model <- "first"
  } else if (!(is.na(epsilon)) && threshold == 0) {
    model <- "decreasing"
  } else if (!(is.na(epsilon)) && threshold == 1) {
    model <- "greedy"
  } else {
    stop("Unknown Model! Plase modify your learning rate function")
  }

  set.seed(rownum)
  # Epsilon-First:
  if (rownum <= threshold) {
    try <- 1
  } else if (rownum > threshold && model == "first") {
    try <- 0
    # Epsilon-Greedy:
  } else if (rownum > threshold && model == "greedy"){
    try <- sample(
      c(1, 0),
      prob = c(epsilon, 1 - epsilon),
      size = 1
    )
    # Epsilon-Decreasing:
  } else if (rownum > threshold && model == "decreasing") {
    try <- sample(
      c(1, 0),
      prob = c(
        1 / (1 + epsilon * rownum),
        epsilon * rownum / (1 + epsilon * rownum)
      ),
      size = 1
    )
  }

  return(try)
}
```

`func_gamma` *Function: Utility Function*

Description

$$U(R) = R^{\gamma}$$

Usage

```
func_gamma(reward, params, ...)
```

Arguments

`reward`	The feedback received by the agent from the environment at trial(t) following the execution of action(a)
`params`	Parameters used by the model's internal functions, see params

`...`	It currently contains the following information; additional information may be added in future package versions.
	• idinfo:
	  – subid
	  – block
	  – trial
	• exinfo: contains information whose column names are specified by the user.
	  – Frame
	  – RT
	  – NetWorth
	  – ...
	• behave: includes the following:
	  – action: the behavior performed by the human in the given trial.
	  – latent: the object updated by the agent in the given trial.
	  – simulation: the actual behavior performed by the agent.

Value

A `NumericVector` of length one representing the subjective value transformed from the objective reward via the utility function.

Body

```
func_gamma <- function(
    reward,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]
```

```
  gamma     <-  params[["gamma"]]

  # Stevens' Power Law
  utility <- sign(reward) * (abs(reward) ^ gamma)

  return(utility)
}
```

`func_zeta` *Function: Decay Rate*

Description

$$W_{new} = W_{old} + \zeta \cdot (W_0 - W_{old})$$

Usage

```
func_zeta(value0, values, reward, params, system, ...)
```

Arguments

`value0`	The initial values for all actions.
`values`	The current expected values for all actions.
`reward`	The feedback received by the agent from the environment at trial(t) following the execution of action(a)
`params`	Parameters used by the model's internal functions, see params
`system`	When the agent makes a decision, is a single system at work, or are multiple systems involved? see system
`...`	It currently contains the following information; additional information may be added in future package versions.

- idinfo:
  - subid
  - block
  - trial
- exinfo: contains information whose column names are specified by the user.
  - Frame
  - RT
  - NetWorth
  - ...
- behave: includes the following:
  - action: the behavior performed by the human in the given trial.
  - latent: the object updated by the agent in the given trial.
  - simulation: the actual behavior performed by the agent.

Value

A `NumericVector` representing the values of unchosen options after decay according to the decay rate.

Body

```
func_zeta <- function(
    value0,
    values,
    reward,
    params,
    ...
){

  list2env(list(...), envir = environment())

  # If you need extra information(...)
  # Column names may be lost(C++), indexes are recommended
  # e.g.
  # Trial  <- idinfo[3]
  # Frame  <- exinfo[1]
  # Action <- behave[1]

  zeta      <-  params[["zeta"]]
  bonus     <-  params[["bonus"]]

  if (reward == 0) {
    decay <- values + zeta * (value0 - values)
  } else if (reward < 0) {
    decay <- values + zeta * (value0 - values) + bonus
  } else if (reward > 0) {
    decay <- values + zeta * (value0 - values) - bonus
  }

  return(decay)
}
```

`MAB` *Simulated Multi-Arm Bandit Dataset*

Description

A simulated multi-armed bandit (MAB) dataset featuring a complex stimulus-response structure. The set of four distinct stimuli (red, blue, yellow, green) is not isomorphic to the set of four available choices (up, down, left, right). Crucially, multiple stimuli may map to the same underlying choice (e.g., Red and Blue both map to 'Up'). This design requires the reinforcement learning model to learn the latent mapping from observable stimuli to the set of potential actions, making it a challenging test case for model fitting.

Format

A data frame with 9000 rows and 12 columns:

Subject Subject ID, an integer ranging from 1 to 30.

Block Block number, an integer ranging from 1 to 6.

Trial Trial number within each block, an integer (1 to 50).

Object_1, Object_2, Object_3, Object_4 Stimulus-response combinations (string) for four objects, formatted as "Color_Direction" (e.g., "Red_Up"). Each column is independently balanced and shuffled.

Reward_1, Reward_2, Reward_3, Reward_4 Reward values for four choice arms (Decks), following the classic Iowa Gambling Task (IGT) structure with adjusted values.

- `Reward_1` (Bad): High gain (+100) with high frequency, mid-sized fine (-250). Long-term net loss.
- `Reward_2` (Bad): High gain (+100) with low frequency, large fine (-1250). Long-term net loss.
- `Reward_3` (Good): Low gain (+50) with high frequency, small fine (-50). Long-term net gain.
- `Reward_4` (Good): Low gain (+50) with low frequency, mid-sized fine (-250). Long-term net gain.

Rewards are balanced at the Block level.

Action The simulated choice made by the subject on that trial (string), randomly sampled from "Up", "Down", "Left", or "Right".

`params` *Model Parameters*

Description

The names of all these parameters are not necessarily fixed. You can define the parameters you need and set their names according to the functions used in your custom model. You must only ensure that the parameter names defined here are consistent with those used in your model's functions, and that their names do not conflict with each other.

Class

`params [List]`

Note

The parameters are divided into three types: `free`, `fixed`, and `constant`. This classification is not mandatory, any parameter can be treated as a free parameter depending on the user's specification. By default, the learning rate `alpha` and the inverse-temperature `beta` are the required free parameters.

Slots

free:

- `alpha [double]`
  Learning Rate `alpha` specifies how aggressively or conservatively the agent adopts the prediction error (the difference between the observed reward and the expected value).
  A value closer to 1 indicates a more aggressive update of the value function, meaning the agent relies more heavily on the current observed reward. Conversely, a value closer to 0 indicates a more conservative update, meaning the agent trusts its previously established expected value more.
- `beta [double]`
  The inverse temperature parameter, `beta`, is a crucial component of the soft-max function. It reflects the extent to which the agent's decision-making relies on the value differences between various available options.
  A higher value of `beta` signifies more rational decision-making; that is, the probability of executing actions with higher expected value is greater. Conversely, a lower `beta` value signifies more stochastic decision-making, where the probability of executing different actions becomes nearly equal, regardless of the differences in their expected values.

fixed:

- `gamma [double]`
  The physical reward received is often distinct from the psychological value perceived by an individual. This concept originates in psychophysics, specifically Stevens' Power Law.
  Note: The default utility function is defined as $y = x^{\gamma}$ and $\gamma = 1$, which assumed that the physical quantity is equivalent to the psychological quantity.
- `delta [double]`
  This parameter represents the weight given to the number of times an option has been selected. Following the Upper Confidence Bound (UCB) algorithm proposed by Sutton and Barto (2018) options that have been selected less frequently should be assigned a higher exploratory bias.
  Note: With the default set to 0.1, a bias value is effectively applied only to options that have never been chosen. Once an action has been executed even a single time, the assigned bias value approaches zero.
- `epsilon [double]`
  This parameter governs the Exploration-Exploitation trade-off and can be used to implement three distinct strategies by adjusting `epsilon` and `threshold`:
  When set to $\epsilon\breve{} greedy$: `epsilon` represents the probability that the agent will execute a random exploratory action throughout the entire experiment, regardless of the estimated value.
  When set to $\epsilon\breve{} decreasing$: The probability of the agent making a random choice decreases as the number of trials increases. The rate of this decay is influenced by `epsilon`.
  By default, `epsilon` is set to `NA`, which corresponds to the $\epsilon\breve{} first$ model. In this model, the agent always selects randomly before a specified trial (`threshold = 1`).
- `zeta [double]`
  Collins and Frank, (2012) doi:10.1111/j.14609568.2011.07980.x proposed that in every trial, not only the chosen option undergoes value updating, but the expected values of unchosen options also decay towards their initial value, due to the constraints of working memory. This specific parameter represents the rate of this decay.
  Note: A larger value signifies a faster decay from the learned value back to the initial value. The default value is set to 0, which assumes that no such working memory system exists.

constant:

- `seed [int]` This seed controls the random choice of actions in the reinforcement learning model when the `sample()` function is called to select actions based on probabilities estimated by the softmax. It is not the seed used by the algorithm package when searching for optimal input parameters. In most cases, there is no need to modify this value; please keep it at the default value of `123`.
- `Q0 [double]`
  This parameter represents the initial value assigned to each action at the start of the Markov Decision Process. As argued by Sutton and Barto (2018), initial values are often set to be optimistic (i.e., higher than all possible rewards) to encourage exploration. Conversely, an overly low initial value might lead the agent to cease exploring other options after receiving the first reward, resulting in repeated selection of the initially chosen action.
  The default value is set to `NA`, which implies that the agent will use the first observed reward as the initial value for that action. When combined with Upper Confidence Bound, this setting ensures that every option is selected at least once, and their first rewards are immediately memorized.
  Note: This is what I consider the reasonable setting. If you think this interpretation unsuitable, you may explicitly set `Q0` to 0 or another optimistic initial value instead.
- `reset [double]`
  If changes may occur between blocks, you can choose whether to reset the learned values for each option. By default, no reset is applied. For example, setting `reset = 0` means that upon entering a new block, the values of all options are reset to 0. In addition, if `Q0` is also set to 0, this implies that the learning rate on the first trial of each block will be 100
- `lapse [double]`
  Wilson and Collins, (2019) doi:10.7554/eLife.49547 introduced the concept of the Lapse Rate, which represents the probability that a subject makes a error (lapse). This parameter ensures that every option has a minimum probability of being chosen, preventing the probability from reaching zero. This is a very reasonable assumption and, crucially, it avoids the numerical instability issue where $\log(P) = \log(0)$ results in `-Inf`.
  Note: The default value here is set to 0.01, meaning every action has at least 1% probability of being executed by the agent. If the paradigm you use have a large number of available actions, a 1% minimum probability for each action might be unreasonable. You can adjust this value to be even smaller.
- `threshold [double]`
  This parameter represents the trial number before which the agent will select completely randomly.
  Note: The default value is set to 1, meaning that only the very first trial involves a purely random choice by the agent.
- `bonus [double]`
  Hitchcock, Kim and Frank, (2025) doi:10.1037/xge0001817 introduced modifications to the working memory model, positing that the value of unchosen options is not merely subject to decay toward the initial value. They suggest that the outcome obtained after selecting an option might, to some extent, provide information about the value of the unchosen options. This information, referred to as a reward bonus, also influences the value update of the unchosen options.
  Note: The default value for this bonus is 0, which assumes that no such bonus value change exists.
- `weight [NumericVector]`

The `weight` parameter governs the policy integration stage. After each cognitive system (e.g., reinforcement learning (RL) and working memory (WM)) calculates action probabilities using a soft-max function based on its internal value estimates, the agent combines these suggestions into a single choice probability.
The default is `1`, which is equivalent to `weight = c(1, 0)`. This represents exclusive reliance on the first system (typically the incremental Reinforcement Learning system).
In a dual-system model (e.g., RL + WM), setting `weight = 0.5` implies that the agent places equal trust in both the long-term RL rewards and the immediate WM memory.

- `capacity [double]`
  This parameter represents the maximum number of stimulus-action associations an individual can actively maintain in working memory $weight = weight_0 * min(1, (capacity/ns))$.
  This parameter determines the extent to which working memory (WM) Q-values are prioritized during decision-making. When the stimulus set size (`ns`) is within the capacity (`capacity`), the model fully relies on the working memory system, resulting in a working memory weight of 1. However, if `ns` exceeds `capacity`, the decision-making process partially integrates Q-values from the reinforcement learning (RL) system.
- `sticky [double]`
  The `sticky` parameter (represented as $kappa$ in Collins, 2025 doi:10.1038/s4156202502340-0) quantifies the extent to which an agent tends to repeat the physical action performed in the previous trial. It captures a form of motor inertia that is fundamentally stimulus-independent.
  Example: Consider a paradigm with four keys (e.g., Up, Down, Left, Right). If an agent pressed "Up" in the previous trial, they might press "Up" again in the current trial, simply due to a reluctance to switch their physical response (i.e., motor stickiness).
  It is imperative that the definition of sticky aligns with the participant's actual physical execution. If a task involves choosing between four bandits (A, B, C, D) displayed on the left or right of a screen, sticky should track the repetition of the physical position (Left or Right) rather than the bandit's identity (A/B/C/D). If your experimental paradigm dissociates the value-updating entities (e.g., bandit IDs) from the physical response dimensions (e.g., spatial locations), you must define the sticky term based on the actual motor response.

Example

```
# TD
params = list(
  free = list(
    alpha = x[1],
    beta = x[2]
  ),
  fixed = list(
    gamma = 1,
    delta = 0.1,
    epsilon = NA_real_,
    zeta = 0
  ),
  constant = list(
    Q0 = NA_real_,
    lapse = 0.01,
    threshold = 1,
    bonus = 0,
```

```
      weight = 1,
      capacity = 0,
      sticky = 0
    )
  )
```

References

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed). MIT press.

Collins, A. G., & Frank, M. J. (2012). How much of reinforcement learning is working memory, not reinforcement learning? A behavioral, computational, and neurogenetic analysis. *European Journal of Neuroscience, 35*(7), 1024-1035. doi:10.1111/j.14609568.2011.07980.x

Wilson, R. C., & Collins, A. G. (2019). Ten simple rules for the computational modeling of behavioral data. *Elife, 8*, e49547. doi:10.7554/eLife.49547

Hitchcock, P. F., Kim, J., Frank, M. J. (2025). How working memory and reinforcement learning interact when avoiding punishment and pursuing reward concurrently. *Journal of Experimental Psychology: General*. doi:10.1037/xge0001817

Collins, A. G. (2025). A habit and working memory model as an alternative account of human reward-based learning. *Nature Human Behaviour*, 1-13. doi:10.1038/s41562025023400

---

plot.multiRL.replay *plot.multiRL.replay*

Description

plot.multiRL.replay

Usage

```
## S3 method for class 'multiRL.replay'
plot(x, y = NULL, model = NULL, param = NULL, ...)
```

Arguments

`x`	multiRL.replay
`y`	NULL
`model`	The name of model that you want to plot
`param`	The name of parameter that you want to plot
`...`	extra

Value

An S3 object of class `ggplot2`

`policy` *Policy of Agent*

Description

The term "policy" in this context is debatable, but the core meaning is whether the model itself acts based on the probabilities it estimates.

Class

`policy [Character]`

Detail

- "On-Policy": The agent converts the expected value of each action into a probability distribution using the soft-max function. It then utilizes a `sample()` function to randomly select an action to execute based on these estimated probabilities. Under this mechanism, actions with higher expected values have a greater likelihood of being selected. Once an action is performed, the feedback received (reward or penalty) is used to update the expected value of that action, which in turn influences the probability of choosing different actions in the future.
- "Off-Policy": The agent directly replicates human behavior. Consequently, in most cases, this ensures that the rewards obtained by the agent in each trial are identical to those obtained by the human. This also results in the value update trajectories for different actions being exactly the same as the trajectories experienced by the human. In this scenario, a previous choice does not influence subsequent value updates. Because all actions are copied from the human, the trajectory of value updates will not diverge due to differences in individual samples. Essentially, in this specific case, the `sample()` step does not exist.

Metaphor

- "On-Policy": The agent completes an examination paper independently and then checks its answers against the ground truth to see if they are correct. If it makes a mistake, it re-attempts the task (adjusting the input parameters). This process repeats until its answers are sufficiently close to the standard answers, or until the degree of similarity can no longer be improved. In other words, the agent has found the optimal parameters within the given model to imitate human behavior as closely as possible.
- "Off-Policy": The agent sees the standard answers to the exam directly. It does not personally complete any of the papers; instead, it acts as an observer trying to understand the underlying logic behind the standard answers. Even if there are a few answers that the agent cannot even understand at all, they will ignore these outliers in order to maximize its overall accuracy.

priors — *Density and Random Function*

Description

Users must specify one of the two function types (`stats::?func`). Either the Density Function (d-func) or the Random Function (r-func)

- Density Function (`stats::dfunc`) represents the prior distribution the free parameters are assumed to follow
- Random Function (`stats::rfunc`) represents the sampling distribution for generating random numbers

Users do not need to memorize when to input the d-func or the r-func; the program will handle the necessary conversion automatically. Since this conversion function relies on regular expressions for string transformation, it is relatively brittle. Users must strictly follow the examples provided below.

Class

`priors [List]`

Density Function

```
# standard format dfunc (Only the numerical values can be modified.)
function(x) {stats::dbeta(x, shape1 = 2, shape2 = 2, log = TRUE)}
function(x) {stats::dexp(x, rate = 1, log = TRUE)}
function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)}
function(x) {stats::dnorm(x, mean = 0.5, sd = 0.1, log = TRUE)}
function(x) {stats::dlnorm(x, meanlog = 0.5, sdlog = 0.1, log = TRUE)}
function(x) {stats::dgamma(x, shape = 2, rate = 3, log = TRUE)}
function(x) {stats::dlogis(x, location = 0, scale = 1, log = TRUE)}
```

Random Function

```
# standard format rfunc  (Only the numerical values can be modified.)
function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)}
function(x) {stats::rexp(n = 1, rate = 1)}
function(x) {stats::runif(n = 1, min = 0, max = 1)}
function(x) {stats::rnorm(n = 1, mean = 0.5, sd = 0.1)}
function(x) {stats::rlnorm(n = 1, meanlog = 0.5, sdlog = 0.1)}
function(x) {stats::rgamma(n = 1, shape = 2, rate = 3)}
function(x) {stats::rlogis(n = 1, location = 0, scale = 1)}
```

Example

```
# TD
params = list(
  free = list(
    alpha = x[1],
    beta = x[2]
  ),
  fixed = list(
    gamma = 1,
    delta = 0.1,
    epsilon = NA_real_,
    zeta = 0
  ),
  constant = list(
    seed = 123,
    Q0 = NA_real_,
    reset = NA_real_,
    lapse = 0.01,
    threshold = 1,
    bonus = 0,
    weight = 1,
    capacity = 0,
    sticky = 0
  )
)

priors = list(
  alpha = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)},
  beta = function(x) {stats::rexp(n = 1, rate = 1)}
)
```

---

process_1_input *multiRL.input*

Description

multiRL.input

Usage

```
process_1_input(
  data,
  colnames = list(),
  funcs = list(),
  params = list(),
  priors,
```

```
  settings = list(),
  ...
)
```

Arguments

`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`funcs`	The functions forming the reinforcement learning model, see funcs
`params`	Parameters used by the model's internal functions, see params
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`...`	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.input`.

`data` A `DataFrame` containing the trial-level raw data.

`colnames` An S4 object of class `multiRL.colnames`, specifying the column names used in the input data.

`features` An S4 object of class `multiRL.features`, containing standardized representations of states and actions transformed from the raw data.

`params` An S4 object of class `multiRL.params`, containing model parameters.

`priors` A `List` specifying prior distributions for free parameters.

`funcs` An S4 object of class `multiRL.funcs`, containing functions used in model.

`settings` An S4 object of class `multiRL.settings`, storing global settings for model estimation.

`elements` A `int` indicating the number of elements within states.

`subid` A `Character` string identifying the subject.

`n_block` A `int` value indicating the number of blocks.

`n_trial` A `int` value indicating the number of trials.

`n_rows` A `int` value indicating the number of rows in the data.

`extra` A `List` containing additional user-defined information.

process_2_behrule *multiRL.behrule*

Description

multiRL.behrule

Usage

```
process_2_behrule(behrule, ...)
```

Arguments

behrule	The agent's implicitly formed internal rule, see behrule
...	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.behrule`.

`cue` A `CharacterVector` containing the cue (state) presented on each trial.

`rsp` A `CharacterVector` containing the set of possible actions available to the agent.

`extra` A `List` containing additional user-defined information.

process_3_record *multiRL.record*

Description

multiRL.record

Usage

```
process_3_record(input, behrule, ...)
```

Arguments

input	multiRL.input
behrule	multiRL.behrule
...	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.record`.

`input` An S4 object of class `multiRL.input`, containing the raw data, column specifications, parameters and ...

`behrule` An S4 object of class `multiRL.behrule`, defining the latent learning rules.

`result` An S4 object of class `multiRL.result`, which is empty for now, storing trial-level outputs of the Markov Decision Process.

`extra` A `List` containing additional user-defined information.

`process_4_output_cpp` *multiRL.output*

Description

multiRL.output

Usage

```
process_4_output_cpp(record, extra)
```

Arguments

`record`	multiRL.record
`extra`	A list of extra information passed from R.

Value

An S4 object of class `multiRL.output`.

`input` An object of class `multiRL.input`, containing the raw data, column specifications, parameters and ...

`behrule` An object of class `multiRL.behrule`, defining the latent learning rules.

`result` An object of class `multiRL.result`, storing trial-level outputs of the Markov Decision Process.

`extra` A `List` containing additional user-defined information.

process_4_output_r *multiRL.output*

Description

multiRL.output

Usage

```
process_4_output_r(record, ...)
```

Arguments

`record`	multiRL.record
`...`	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.output`.

- `input` An object of class `multiRL.input`, containing the raw data, column specifications, parameters and ...
- `behrule` An object of class `multiRL.behrule`, defining the latent learning rules.
- `result` An object of class `multiRL.result`, storing trial-level outputs of the Markov Decision Process.
- `extra` A `List` containing additional user-defined information.

process_5_metric *multiRL.metric*

Description

multiRL.metric

Usage

```
process_5_metric(output, ...)
```

Arguments

`output`	multiRL.output
`...`	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.metric`.

`input` An S4 object of class `multiRL.input`, containing the raw data, column specifications, parameters and ...

`behrule` An S4 object of class `multiRL.behrule`, defining the latent learning rules.

`result` An S4 object of class `multiRL.result`, storing trial-level outputs of the Markov Decision Process.

`sumstat` An S4 object of class `multiRL.sumstat`, providing summary statistics across different estimation methods.

`extra` A `List` containing additional user-defined information.

---

rcv_d — *Step 2: Generating fake data for parameter and model recovery*

Description

Step 2: Generating fake data for parameter and model recovery

Usage

```
rcv_d(
  estimate,
  data,
  colnames,
  behrule,
  id = NULL,
  models,
  funcs = NULL,
  priors = NULL,
  settings = NULL,
  algorithm,
  lowers,
  uppers,
  control,
  ...
)
```

Arguments

`estimate`	Estimate method that you want to use, see estimate
`data`	A data frame in which each row represents a single trial, see data
`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule

id	The ID of the subject whose experimental structure (e.g., trial order) will be used as the template for generating all simulated data. Defaults to the first subject found in the input data.
models	Reinforcement Learning Models
funcs	The functions forming the reinforcement learning model, see funcs
priors	Prior probability density function of the free parameters, see priors
settings	Other model settings, see settings
algorithm	Algorithm packages that multiRL supports, see algorithm
lowers	Lower bound of free parameters in each model.
uppers	Upper bound of free parameters in each model.
control	Settings manage various aspects of the iterative process, see control
...	Additional arguments passed to internal functions.

Value

An S3 object of class `multiRL.recovery`.

`simulate` A `List` containing, for each model, the parameters used to simulate the data.

`recovery` A `List` containing, for each model, the parameters estimated as optimal by the algorithm.

Example

```
# recovery
recovery.MLE <- multiRL::rcv_d(
  estimate = "MLE",

  data = multiRL::TAB,
  colnames = list(
    object = c("L_choice", "R_choice"),
    reward = c("L_reward", "R_reward"),
    action = "Sub_Choose"
  ),
  behrule = list(
    cue = c("A", "B", "C", "D"),
    rsp = c("A", "B", "C", "D")
  ),
  id = 1,

  models = list(multiRL::TD, multiRL::RSTD, multiRL::Utility),
  priors = list(
    list(
      alpha = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)},
      beta = function(x) {stats::rexp(n = 1, rate = 1)}
    ),
    list(
```

```
      alphaN = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)},
      alphaP = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)},
      beta = function(x) {stats::rexp(n = 1, rate = 1)}
    ),
    list(
      alpha = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)},
      beta = function(x) {stats::rexp(n = 1, rate = 1)},
      gamma = function(x) {stats::rbeta(n = 1, shape1 = 2, shape2 = 2)}
    )
  ),
  settings = list(name = c("TD", "RSTD", "Utility")),

  algorithm = "NLOPT_GN_MLSL",
  lowers = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
  uppers = list(c(1, 5), c(1, 1, 5), c(1, 5, 1)),
  control = list(core = 10, sample = 100, iter = 100)
 )
```

`rpl_e` *Step 4: Replaying the experiment with optimal parameters*

Description

Step 4: Replaying the experiment with optimal parameters

Usage

```
rpl_e(
  result,
  free_params = NULL,
  data,
  colnames,
  behrule,
  ids = NULL,
  models,
  funcs = NULL,
  priors = NULL,
  settings = NULL,
  ...
)
```

Arguments

`result`	Result from `rcv_d` or `fit_p`
`free_params`	In order to prevent ambiguity regarding the free parameters, their names can be explicitly defined by the user.
`data`	A data frame in which each row represents a single trial, see data

`colnames`	Column names in the data frame, see colnames
`behrule`	The agent's implicitly formed internal rule, see behrule
`ids`	The Subject ID of the participant whose data needs to be fitted.
`models`	Reinforcement Learning Models
`funcs`	The functions forming the reinforcement learning model, see funcs
`priors`	Prior probability density function of the free parameters, see priors
`settings`	Other model settings, see settings
`...`	Additional arguments passed to internal functions.

Value

An S3 object of class `multiRL.replay`. A `List` containing, for each subject and each fitted model, the estimated optimal parameters, along with the resulting `multiRL.model` and `multiRL.summary` objects obtained by replaying the model with those parameters.

Example

```
# info
data = multiRL::TAB
colnames = list(
  object = c("L_choice", "R_choice"),
  reward = c("L_reward", "R_reward"),
  action = "Sub_Choose"
)
behrule = list(
  cue = c("A", "B", "C", "D"),
  rsp = c("A", "B", "C", "D")
)

replay.recovery <- multiRL::rpl_e(
  result = recovery.MLE,

  data = data,
  colnames = colnames,
  behrule = behrule,

  models = list(multiRL::TD, multiRL::RSTD, multiRL::Utility),
  settings = list(name = c("TD", "RSTD", "Utility")),

  omit = c("data", "funcs")
)

replay.fitting <- multiRL::rpl_e(
  result = fitting.MLE,

  data = data,
  colnames = colnames,
```

```
    behrule = behrule,

    models = list(multiRL::TD, multiRL::RSTD, multiRL::Utility),
    settings = list(name = c("TD", "RSTD", "Utility")),

    omit = c("funcs")
  )
```

RSTD *Risk Sensitive Model*

Description

Learning Rate: α

$$Q_{new} = Q_{old} + \alpha_- \cdot (R - Q_{old}), R < Q_{old}$$
$$Q_{new} = Q_{old} + \alpha_+ \cdot (R - Q_{old}), R \geq Q_{old}$$

Inverse Temperature: β

$$P_t(a) = \frac{\exp(\beta \cdot Q_t(a))}{\sum_{i=1}^{k} \exp(\beta \cdot Q_t(a_i))}$$

Usage

```
RSTD(params)
```

Arguments

`params`	Parameters used by the model's internal functions, see params

Value

Depending on the `mode` and `estimate` defined in the runtime environment, the corresponding outputs for different estimation methods are produced, such as a single log-likelihood value or summary statistics.

Body

```
RSTD <- function(params){

  params <- list(
    free = list(alphaN = params[1], alphaP = params[2], beta = params[3])
  )

  multiRL.model <- multiRL::run_m(
    data = data,
```

```
    behrule = behrule,
    colnames = colnames,
    params = params,
    funcs = funcs,
    priors = priors,
    settings = settings
  )

  assign(x = "multiRL.model", value = multiRL.model, envir = multiRL.env)
  return(.return_result(multiRL.model))
}
```

---

run_m *Step 1: Building reinforcement learning model*

Description

Step 1: Building reinforcement learning model

Usage

```
run_m(
  data,
  colnames = list(),
  behrule = list(),
  funcs = list(),
  params = list(),
  priors = list(),
  settings = list(),
  engine = "Cpp",
  ...
)
```

Arguments

data	A data frame in which each row represents a single trial, see data
colnames	Column names in the data frame, see colnames
behrule	The agent's implicitly formed internal rule, see behrule
funcs	The functions forming the reinforcement learning model, see funcs
params	Parameters used by the model's internal functions, see params
priors	Prior probability density function of the free parameters, see priors
settings	Other model settings, see settings
engine	Specifies whether the core Markov Decision Process (MDP) update loop is executed in C++ or in R.
...	Additional arguments passed to internal functions.

Value

An S4 object of class `multiRL.model`.

`input` An S4 object of class `multiRL.input`, containing the raw data, column specifications, parameters and ...

`behrule` An S4 object of class `multiRL.behrule`, defining the latent learning rules.

`result` An S4 object of class `multiRL.result`, storing trial-level outputs of the Markov Decision Process.

`sumstat` An S4 object of class `multiRL.sumstat`, providing summary statistics across different estimation methods.

`extra` A `List` containing additional user-defined information.

Examples

```
multiRL.model <- multiRL::run_m(
  data = multiRL::TAB[multiRL::TAB[, "Subject"] == 1, ],
  behrule = list(
    cue = c("A", "B", "C", "D"),
    rsp = c("A", "B", "C", "D")
  ),
  colnames = list(
    subid = "Subject", block = "Block", trial = "Trial",
    object = c("L_choice", "R_choice"),
    reward = c("L_reward", "R_reward"),
    action = "Sub_Choose",
    exinfo = c("Frame", "NetWorth", "RT")
  ),
  params = list(
    free = list(
      alpha = 0.5,
      beta = 0.5
    ),
    fixed = list(
      gamma = 1,
      delta = 0.1,
      epsilon = NA_real_,
      zeta = 0
    ),
    constant = list(
      seed = 123,
      Q0 = NA_real_,
      reset = NA_real_,
      lapse = 0.01,
      threshold = 1,
      bonus = 0,
      weight = 1,
      capacity = 0,
      sticky = 0
    )
  ),
```

```
  priors = list(
    alpha = function(x) {stats::dbeta(x, shape1 = 2, shape2 = 2, log = TRUE)},
    beta = function(x) {stats::dexp(x, rate = 1, log = TRUE)}
  ),
  settings = list(
    name = "TD",
    mode = "fitting",
    estimate = "MLE",
    policy = "off",
    system = c("RL", "WM")
 ),
 engine = "R"
)

multiRL.summary <- multiRL::summary(multiRL.model)
```

---

settings *Settings of Model*

Description

The `settings` argument is responsible for defining the model's name, the estimation method, and other configurations.

Class

`settings [List]`

Slots

- `name [Character]`

  The name of model.

- `mode [Character]`

  There are two modes: `"fitting"` and `"simulating"`. In most cases, users do not need to explicitly specify the value of this slot, as the program will set it automatically.

  Typically, the `"fitting"` mode is used when executing `fit_p`, while the `"simulating"` mode is used when executing `rcv_d`.

- `estimate [Character]`

  The package supports four estimation methods: Maximum Likelihood Estimation (MLE), Maximum A Posteriori Estimation (MAP), Approximate Bayesian Computation (ABC), and Recurrent Neural Network (RNN). Generally, users no longer need to specify the estimation method in the settings object. This slot has been moved to an argument within the main functions, `rcv_d` and `fit_p`. For details, please refer to the documentation for estimate.

- `policy [Character]`

  The naming of this slot as policy is still under consideration.

  Colloquially, `policy = "on"` means the agent selects an option based on its estimated probability and then updates the value of the chosen option.

  Conversely, `policy = "off"` means the agent directly mimics human behavior, solely using its estimated probability and the human's choice to calculate the likelihood.

  For details, please refer to the documentation for policy.

- `system [Character]`

  In decision-making paradigms, multiple systems may operate jointly to influence human decisions. These systems can include a reinforcement learning system, as well as working memory, and even habitual choice tendencies.

  If `system = "RL"`, the learning process follows the Rescorla-Wagner (RW) model using a learning rate less than 1, representing a slow, incremental value update system.

  If `system = "WM"`, the process still follows the Rescorla-Wagner (RW) model but with a fixed learning rate of 1, functioning as a pure memory system that immediately updates an option's value.

  If `system = c("RL", "WM")`, the agent maintains two distinct Q-tables, one for reinforcement learning (RL) and one for working memory (WM), during the decision-making process, integrating their values based on the provided `weight` to determine the final choice.

  For details, please refer to the documentation for system.

Example

```
# model settings
settings = list(
  name = "TD",
  mode = "fitting",
  estimate = "MLE",
  policy = "off",
  system = "RL"
)
```

---

summary,multiRL.model-method
summary

Description

summary

Usage

```
## S4 method for signature 'multiRL.model'
summary(object, ...)
```

Arguments

`object`	multiRL.model.
`...`	...

Value

multiRL.summary

`system` *Cognitive Processing System*

Description

In a Markov Decision Process, an agent may not update only a single Q-value table. In other words, the process may not be governed by a single cognitive processing system, but rather by a weighted combination of multiple cognitive systems. Specifically, each cognitive processing system updates its own Q-value table and, based on that table, derives the probabilities of executing each action on a given trial. The agent then combines the action-selection probabilities provided by each cognitive system using weights to obtain the final probability of executing each action.

Class

`system [Character]`

Detail

- Reinforcement Learning: An incremental cognitive processing system that integrates reward history over long timescales to build stable action-value representations through prediction errors. It is robust but slow to adapt to sudden changes.
- Working Memory: A rapid-acquisition cognitive processing system that allows for near-instantaneous updating of stimulus-response associations. However, its contribution is strictly constrained by limited storage capacity and is highly susceptible to decay over time or interference from intervening trials.

Example

- `system = "RL"`: A single-system model based on incremental Reinforcement Learning (RL). The agent updates option values using a learning rate (alpha) typically less than 1, representing a slow, integrative process linked to corticostriatal circuitry.
- `system = "WM"`: A single-system model representing Working Memory (WM). Unlike RL, this system has the capacity to instantly update values with a fixed learning rate of 1, effectively "remembering" the most recent outcome for each stimulus.

- `system = c("RL", "WM")`: A hybrid model where Reinforcement Learning (RL) and Working Memory (WM) systems operate in parallel, maintaining two distinct Q-value tables. The final decision is a weighted integration of both systems' choice probabilities. The contribution of Working Memory (WM) is constrained by its `capacity`; if the stimulus set size exceeds `capacity`, the agent's reliance shifts toward the Reinforcement Learning (RL) system as the Working Memory (WM) reliability diminishes. See `capacity` in params for details.

  If one assumes that multiple cognitive processing systems are involved in the Markov Decision Process, their relative influence can be controlled by assigning `weights` to each system. See `weight` in params for details.

References

Collins, A. G., & Frank, M. J. (2012). How much of reinforcement learning is working memory, not reinforcement learning? A behavioral, computational, and neurogenetic analysis. *European Journal of Neuroscience, 35*(7), 1024-1035. doi:10.1111/j.14609568.2011.07980.x

---

`TAB` *Group 2 from Mason et al. (2024)*

Description

This dataset originates from Experiment 2 of Mason et al. (2024), titled "Rare and extreme outcomes in risky choice" (doi:10.3758/s1342302302415x). The raw data is publicly available on the Open Science Framework (OSF) at `https://osf.io/hy3q4/`. For the purposes of this package, we've performed basic cleaning and preprocessing of the original dataset.

Format

A data frame with 45000 rows and 11 columns:

Subject Subject ID, an integer (total of 143).

Block Block number, an integer (1 to 6).

Trial Trial number, an integer (1 to 60).

L_choice Left choice, a character indicating the option presented. The possible options are:

- A: 100% gain 36.
- B: 90% gain 40 and 10% gain 0.
- C: 100% lose 36.
- D: 90% lose 40 and 10% lose 0.

R_choice Right choice, a character indicating the option presented. The possible options are:

- A: 100% gain 36.
- B: 90% gain 40 and 10% gain 0.
- C: 100% lose 36.
- D: 90% lose 40 and 10% lose 0.

L_reward Reward associated with the left choice.

R_reward Reward associated with the right choice.

Sub_Choose The chosen option, either `L_choice` or `R_choice`.

Frame Type of frame, a character string (e.g., "Gain", "Loss", "Catch").

NetWorth The participant's net worth at the end of each trial.

RT The participant's reaction time (in milliseconds) for each trial.

`TD` *Temporal Differences Model*

Description

Learning Rate: α

$$Q_{new} = Q_{old} + \alpha \cdot (R - Q_{old})$$

Inverse Temperature: β

$$P_t(a) = \frac{\exp(\beta \cdot Q_t(a))}{\sum_{i=1}^{k} \exp(\beta \cdot Q_t(a_i))}$$

Usage

```
TD(params)
```

Arguments

`params`	Parameters used by the model's internal functions, see params

Value

Depending on the `mode` and `estimate` defined in the runtime environment, the corresponding outputs for different estimation methods are produced, such as a single log-likelihood value or summary statistics.

Body

```
TD <- function(params){

  params <- list(
    free = list(alpha = params[1], beta = params[2])
  )

  multiRL.model <- multiRL::run_m(
    data = data,
    behrule = behrule,
    colnames = colnames,
```

```
    params = params,
    funcs = funcs,
    priors = priors,
    settings = settings
  )

  assign(x = "multiRL.model", value = multiRL.model, envir = multiRL.env)
  return(.return_result(multiRL.model))
}
```

Utility *Utility Model*

Description

Learning Rate: α

$$Q_{new} = Q_{old} + \alpha \cdot (U(R) - Q_{old})$$

Inverse Temperature: β

$$P_t(a) = \frac{\exp(\beta \cdot Q_t(a))}{\sum_{i=1}^{k} \exp(\beta \cdot Q_t(a_i))}$$

Stevens' Power-law Exponent: γ

$$U(R) = R^{\gamma}$$

Usage

```
Utility(params)
```

Arguments

`params`	Parameters used by the model's internal functions, see params

Value

Depending on the `mode` and `estimate` defined in the runtime environment, the corresponding outputs for different estimation methods are produced, such as a single log-likelihood value or summary statistics.

Body

```
Utility <- function(params){

  params <- list(
    free = list(alpha = params[1], beta = params[2], gamma = params[3])
  )

  multiRL.model <- multiRL::run_m(
    data = data,
    behrule = behrule,
    colnames = colnames,
    params = params,
    funcs = funcs,
    priors = priors,
    settings = settings
  )

  assign(x = "multiRL.model", value = multiRL.model, envir = multiRL.env)
  return(.return_result(multiRL.model))
}
```

Index

algorithm, 3, *13*, *16–18*, *22, 23*, *52*

behrule, 4, *11, 12*, *15*, *17–20*, *22, 23*, *48*, *51*, *54*, *56*

colnames, 5, *11, 12*, *15*, *17–20*, *22, 23*, *47*, *51*, *54*, *56*
control, 6, *11, 12*, *16–23*, *52*

data, 9, *11, 12*, *15*, *17–20*, *22, 23*, *47*, *51*, *53*, *56*

engine_ABC, 10
engine_RNN, 11
estimate, 12, *22, 23*, *51*, *58*
estimate_0_ENV, 15
estimate_1_LBI, 16
estimate_1_MAP, 16
estimate_1_MLE, 18
estimate_2_ABC, 19
estimate_2_RNN, 20
estimate_2_SBI, 21
estimation_methods, 21

fit_p, 23
func_alpha, 28
func_beta, 30
func_delta, 32
func_epsilon, 33
func_gamma, 35
func_zeta, 37
funcs, *11, 12*, *15*, *17–20*, *22, 23*, 24, *47*, *52*, *54*, *56*

MAB, 38

params, *28*, *30*, *32*, *34*, *36, 37*, 39, *47*, *55, 56*, *61–63*
plot.multiRL.replay, 43
policy, 44, *59*
priors, *11, 12*, *15*, *17–23*, 45, *47*, *52*, *54*, *56*

process_1_input, 46
process_2_behrule, 48
process_3_record, 48
process_4_output_cpp, 49
process_4_output_r, 50
process_5_metric, 50

rcv_d, 51
rpl_e, 53
RSTD, 55
run_m, 56

settings, *11, 12*, *15*, *17–20*, *22, 23*, *47*, *52*, *54*, *56*, 58
summary,multiRL.model-method, 59
system, *28*, *30*, *37*, *59*, 60

TAB, 61
TD, 62

Utility, 63