Package: binaryRL (via r-universe)

October 9, 2026

Version 0.9.11

Title Reinforcement Learning Tools for Two-Alternative Forced Choice Tasks

Description Tools for building Rescorla-Wagner Models for Two-Alternative Forced Choice tasks, commonly employed in psychological research. Most concepts and ideas within this R package are referenced from Sutton and Barto (2018) <ISBN:9780262039246>. The package allows for the intuitive definition of RL models using simple if-else statements and three basic models built into this R package are referenced from Niv et al. (2012) <doi:10.1523/JNEUROSCI.5498-10.2012>. Our approach to constructing and evaluating these computational models is informed by the guidelines proposed in Wilson & Collins (2019) <doi:10.7554/eLife.49547>. Example datasets included with the package are sourced from the work of Mason et al. (2024) <doi:10.3758/s13423-023-02415-x>.

Maintainer YuKi <hmz1969a@gmail.com>

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

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

License GPL-3

Encoding UTF-8

LazyData TRUE

ByteCompile TRUE

RoxygenNote 7.3.3

Depends R (>= 4.0.0)

Imports Rcpp, compiler, future, doFuture, foreach, doRNG, progressr

LinkingTo Rcpp

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

Repository https://yuki-961004.r-universe.dev

Date/Publication 2026-03-13 09:13:58 UTC

RemoteUrl https://github.com/yuki-961004/binaryrl

RemoteRef HEAD

RemoteSha 6078e067425a24fac40a993f1eb375b44e2fe770

Contents

fit_p . . . 2
func_epsilon . . . 9
func_eta . . . 12
func_gamma . . . 15
func_logl . . . 18
func_pi . . . 21
func_tau . . . 23
Mason_2024_G1 . . . 27
Mason_2024_G2 . . . 28
optimize_para . . . 29
rcv_d . . . 34
recovery_data . . . 41
rpl_e . . . 45
RSTD . . . 47
run_m . . . 48
simulate_list . . . 56
summary.binaryRL . . . 58
TD . . . 59
Utility . . . 60

Index **61**

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

Description

This function is designed to fit the optimal parameters of black-box functions (models) to real-world data. Provided that the black-box function adheres to the specified interface (demo: `TD`, `RSTD`, `Utility`) , this function can employ the various optimization algorithms detailed below to find the best- fitting parameters for your model.

The function provides several optimization algorithms:

- 1. L-BFGS-B (from `stats::optim`)
- 2. Simulated Annealing (`GenSA::GenSA`)
- 3. Genetic Algorithm (`GA::ga`)
- 4. Differential Evolution (`DEoptim::DEoptim`)
- 5. Particle Swarm Optimization (`pso::psoptim`)

- 6. Bayesian Optimization (`mlrMBO::mbo`)
- 7. Covariance Matrix Adapting Evolutionary Strategy (`cmaes::cma_es`)
- 8. Nonlinear Optimization (`nloptr::nloptr`)

For more information, please refer to the homepage of this package: https://yuki-961004.github.io/binaryRL/

Usage

```
fit_p(
  estimate = "MLE",
  policy = "off",
  data,
  id = NULL,
  n_trials = NULL,
  funcs = NULL,
  model_name = c("TD", "RSTD", "Utility"),
  fit_model = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  priors = NULL,
  lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
  upper = list(c(1, 5), c(1, 1, 5), c(1, 1, 5)),
  initial_params = NA,
  initial_size = 50,
  tolerance = 0.001,
  seed = 123,
  iteration_i = 10,
  iteration_g = 0,
  nc = 1,
  algorithm
)
```

Arguments

`estimate` [string]

Estimation method. Can be either `"MLE"` or `"MAP"`.

- **Maximum Likelihood Estimation** `"MLE"`: (Default): This method finds the parameter values that maximize the log-likelihood of the data. A higher log-likelihood indicates that the parameters provide a better explanation for the observed human behavior. In other words, data simulated using these parameters would most closely resemble the actual human data. This method does not consider any prior information about the parameters.
- **Maximum A Posteriori Estimation** `"MAP"`: This method finds the parameter values that maximize the posterior probability. It is an iterative process based on the Expectation-Maximization (EM) framework.
  - **Initialization**: The process begins by assuming a uniform distribution as the prior for each parameter, making the initial log-prior zero. The first optimization is thus equivalent to MLE.

- **Iteration**: After finding the best parameters for all subjects, the algorithm assesses the actual distribution of each parameter and fits a normal distribution to it. This fitted distribution becomes the new empirical prior.
- **Re-estimation**: The parameters are then re-optimized to maximize the updated posterior probability.
- **Convergence**: This cycle repeats until the posterior probability converges or the maximum number of iterations is reached.

Using this method requires that the `priors` argument be specified to define the initial prior distributions.

default: `estimate = "MLE"`

`policy` [string]

Specifies the learning policy to be used. This determines how the model updates action values based on observed or simulated choices. It can be either `"off"` or `"on"`.

- **Off-Policy (Q-learning):** This is the most common approach for modeling reinforcement learning in Two-Alternative Forced Choice (TAFC) tasks. In this mode, the model's goal is to learn the underlying value of each option by observing the human participant's behavior. It achieves this by consistently updating the value of the option that the human actually chose. The focus is on understanding the value representation that likely drove the participant's decisions.
- **On-Policy (SARSA):** In this mode, the target policy and the behavior policy are identical. The model first computes the selection probability for each option based on their current values. Critically, it then uses these probabilities to sample its own action. The value update is then performed on the action that the model itself selected. This approach focuses more on directly mimicking the stochastic choice patterns of the agent, rather than just learning the underlying values from a fixed sequence of actions.

default: `policy = "off"`

`data` [data.frame]

This data should include the following mandatory columns:

- `sub` "Subject"
- `time_line` "Block" "Trial"
- `L_choice` "L_choice"
- `R_choice` "R_choice"
- `L_reward` "L_reward"
- `R_reward` "R_reward"
- `sub_choose` "Sub_Choose"

`id` [CharacterVector]

A vector specifying the subject ID(s) for which parameters should be fitted. The function will process only the subjects provided in this vector.

To fit all subjects, you can either explicitly set the argument as `id = unique(data$Subject)` or leave it as the default (`id = NULL`). Both approaches will direct the function to fit parameters for every unique subject in the dataset.

It is strongly recommended to avoid using simple numeric sequences like `id = 1:4`. This practice can lead to errors if subject IDs are stored as strings (e.g., subject four is stored as "004") or are not sequentially numbered.

default: `id = NULL`

`n_trials` [integer]

Represents the total number of trials a single subject experienced in the experiment. If this parameter is kept at its default value of `NULL`, the program will automatically detect how many trials a subject experienced from the provided data. This information is primarily used for calculating model fit statistics such as AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion).

default: `n_trials = NULL`

`funcs` [CharacterVector]

A character vector containing the names of all user-defined functions required for the computation. When parallel computation is enabled (i.e., `nc > 1`), user-defined models and their custom functions might not be automatically accessible within the parallel environment.

Therefore, if you have created your own reinforcement learning model that modifies the package's default six default functions (default functions: `util_func = func_gamma`, `rate_func = func_eta`, `expl_func = func_epsilon`, `bias_func = func_pi`, `prob_func = func_tau`, `loss_func = func_logl`), you must explicitly provide the names of your custom functions as a vector here.

`model_name` [List]

The name of fit modals

e.g. `model_name = c("TD", "RSTD", "Utility")`

`fit_model` [List]

A collection of functions applied to fit models to the data.

e.g. `fit_model = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility)`

`priors` [List]

A list specifying the prior distributions for the model parameters. This argument is mandatory when using `estimate = "MAP"`. There are two primary scenarios for its use:

1. Static MAP Estimation (Non-Hierarchical) This approach is used when you have a strong, pre-defined belief about the parameter priors and do not want the model to update them iteratively.

Configuration:
- Set `estimate = "MAP"`.
- Provide `pirors` defining probability density function of free parameters
- Keep `iteration_g = 0` (the default).

Behavior: The algorithm maximizes the posterior probability based solely on your specified priors. It will not use the EM (Expectation-Maximization) framework to learn new priors from the data.

2. Hierarchical Bayesian Estimation via EM This approach is used to let the model learn the group-level (hierarchical) prior distributions directly from the data.

Configuration: • Set `estimate = "MAP"`.

- Specify a weak or non-informative initial prior, such as a uniform distribution for all free parameters.
- Set `iteration_g` to a value greater than 0.

Behavior: With a uniform prior, the initial log-posterior equals the log-likelihood, making the first estimation step equivalent to MLE. The algorithm then initiates the EM procedure: it iteratively assesses the actual parameter distribution across all subjects and updates the group-level priors. This cycle continues until the posterior converges or `iteration_g` is reached.

default: `priors = NULL`

`lower` [List]

The lower bounds for model fit models

e.g. `lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0))`

`upper` [List]

The upper bounds for model fit models

e.g. `upper = list(c(1, 5), c(1, 1, 5), c(1, 1, 5))`

`initial_params` [NumericVector]

Initial values for the free parameters that the optimization algorithm will search from. These are primarily relevant when using algorithms that require an explicit starting point, such as `L-BFGS-B`. If not specified, the function will automatically generate initial values close to zero.

default: `initial_params = NA`.

`initial_size` [integer]

This parameter corresponds to the population size in genetic algorithms (`GA`). It specifies the number of initial candidate solutions that the algorithm starts with for its evolutionary search. This parameter is only required for optimization algorithms that operate on a population, such as `GA` or `DEoptim`.

default: `initial_size = 50`.

`tolerance` [double]

Convergence threshold for MAP estimation. If the change in log posterior probability between iterations is smaller than this value, the algorithm is considered to have converged and the program will stop.

default: `tolerance = 0.001`

`seed` [integer]

Random seed. This ensures that the results are reproducible and remain the same each time the function is run.

default: `seed = 123`

`iteration_i` [integer]

The number of iterations the optimization algorithm will perform when searching for the best-fitting parameters during the fitting phase. A higher number of iterations may increase the likelihood of finding a global optimum but also increases computation time.

default: `iteration_i = 10`.

`iteration_g` [integer]

The maximum number of iterations for the Expectation-Maximization (EM) based MAP estimation. The algorithm will stop once this iteration count is reached, even if the change in the log-posterior value has not yet fallen below the `tolerance` threshold.

default: `iteration_g = 0`.

`nc` [integer]

Number of cores to use for parallel processing. Since fitting optimal parameters for each subject is an independent task, parallel computation can significantly speed up the fitting process:

- **nc = 1**: The fitting proceeds sequentially. Parameters for one subject are fitted completely before moving to the next subject.
- **nc > 1**: The fitting is performed in parallel across subjects. For example, if `nc = 4`, the algorithm will simultaneously fit data for four subjects. Once these are complete, it will proceed to fit the next batch of subjects (e.g., subjects 5-8), and so on, until all subjects are processed.

`default: nc = 1`

`algorithm` [string]

Choose an algorithm package from `L-BFGS-B`, `GenSA`,`GA`,`DEoptim`,`PSO`, `Bayesian`, `CMA-ES`.

In addition, any algorithm from the `nloptr` package is also supported. If your chosen `nloptr` algorithm requires a local search, you need to input a character vector. The first element represents the algorithm used for global search, and the second element represents the algorithm used for local search.

e.g. `algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")`

Value

The optimal parameters found by the algorithm for each subject, along with the model fit calculated using these parameters. This is returned as an object of class `binaryRL` containing results for all subjects with all models.

Note

While both `fit_p` and `rcv_d` utilize the same underlying `optimize_para` function to find optimal parameters, they play distinct and sequential roles in the modeling pipeline.

The key differences are as follows:

1. **Purpose and Data Source**: `rcv_d` should always be performed before `fit_p`. Its primary role is to validate a model's stability by fitting it to synthetic data generated by the model itself. This process, known as parameter recovery, ensures the model is well-behaved. In contrast, `fit_p` is used in the subsequent stage to fit the validated model to real experimental data.
2. **Estimation Method**: `rcv_d` does not include an `estimate` argument. This is because the synthetic data is generated from known "true" parameters, which are drawn from pre-defined distributions (typically uniform for most parameters and exponential for the inverse temperature). Since the ground truth is known, a hierarchical estimation (MAP) is not applicable. The

`fit_p` function, however, requires this argument to handle real data where the true parameters are unknown.

3. **Policy Setting**: In `fit_p`, the `policy` setting has different effects: "on-policy" is better for learning choice patterns, while "off-policy" yields more accurate parameter estimates. For `rcv_d`, the process defaults to an "off-policy" approach because its main objectives are to verify if the true parameters can be accurately recovered and to assess whether competing models are distinguishable, tasks for which off-policy estimation is more suitable.

Examples

```
## Not run:
comparison <- binaryRL::fit_p(
  data = binaryRL::Mason_2024_G2,
#+-----------------------------------------------------------------------------+#
#|---------------------------- black-box function ----------------------------|#
  funcs = c("your_funcs"),
  estimate = c("MLE", "MAP"),
  policy = c("off", "on"),
  model_name = c("TD", "RSTD", "Utility"),
#|-------------------------------- fit models --------------------------------|#
  fit_model = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  priors = list(
    list(
      eta = function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)},
      tau = function(x) {stats::dexp(x, rate = 1, log = TRUE)}
    ),
    list(
      eta = function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)},
      eta = function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)},
      tau = function(x) {stats::dexp(x, rate = 1, log = TRUE)}
    ),
    list(
      eta = function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)},
      gamma = function(x) {stats::dunif(x, min = 0, max = 1, log = TRUE)},
      tau = function(x) {stats::dexp(x, rate = 1, log = TRUE)}
    )
  ),
#|---------------------------------- bound ----------------------------------|#
  lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
  upper = list(c(1, 10), c(1, 1, 10), c(1, 1, 10)),
#|---------------------------- iteration number -----------------------------|#
  iteration_i = 10,
  iteration_g = 10,
#|-------------------------------- algorithms --------------------------------|#
  nc = 1,                 # <nc > 1>: parallel computation across subjects
  # Base R Optimization
  #algorithm = "L-BFGS-B"  # Gradient-Based (stats)
#|-----------------------------------------------------------------------------|#
  # Specialized External Optimization
  #algorithm = "GenSA"    # Simulated Annealing (GenSA)
  #algorithm = "GA"       # Genetic Algorithm (GA)
  #algorithm = "DEoptim"  # Differential Evolution (DEoptim)
```

```
  #algorithm = "PSO"      # Particle Swarm Optimization (pso)
  #algorithm = "Bayesian" # Bayesian Optimization (mlrMBO)
  #algorithm = "CMA-ES"   # Covariance Matrix Adapting (cmaes)
#|-------------------------------------------------------------------------------|#
  # Optimization Library (nloptr)
  algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")
#|------------------------------ algorithms ------------------------------|#
################################################################################
)

result <- dplyr::bind_rows(comparison)

# Ensure the output directory exists before writing
if (!dir.exists("../OUTPUT")) {
  dir.create("../OUTPUT", recursive = TRUE)
}

write.csv(result, "../OUTPUT/result_comparison.csv", row.names = FALSE)

## End(Not run)
```

`func_epsilon` *Function: Epsilon Related*

Description

The exploration strategy parameters are `threshold`, `epsilon`, and `lambda`.

- **Epsilon-first:** Used when only `threshold` is set. Subjects choose randomly for trials less than `threshold` and by value for trials greater than ‘`threshold`.
- **Epsilon-greedy:** Used if `threshold` is default (1) and `epsilon` is set. Subjects explore with probability `epsilon` throughout the experiment.
- **Epsilon-decreasing:** Used if `threshold` is default (1), and `lambda` is set. In this strategy, the probability of random choice (exploration) decreases as trials increase. The parameter `lambda` controls the rate at which this probability declines with each trial.

Usage

```
func_epsilon(
  i,
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  var1 = NA,
```

```
  var2 = NA,
  threshold = 1,
  epsilon = NA,
  lambda = NA,
  alpha,
  beta
)
```

Arguments

i	The current row number.
L_freq	The frequency of left option appearance
R_freq	The frequency of right option appearance
L_pick	The number of times left option was picked
R_pick	The number of times left option was picked
L_value	The value of the left option
R_value	The value of the right option
var1	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. `default: var1 = "Extra_Var1"`
var2	[character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model. `default: var2 = "Extra_Var2"`

threshold	[integer] Controls the initial exploration phase in the **epsilon-first** strategy. This is the number of early trials where the subject makes purely random choices, as they haven't yet learned the options' values. For example, `threshold = 20` means random choices for the first 20 trials. For **epsilon-greedy** or **epsilon-decreasing** strategies, `threshold` should be kept at its default value.
	$$P(x) = \begin{cases} \text{trial} \leq \text{threshold}, & x = 1 \text{ (random choosing)} \\ \text{trial} > \text{threshold}, & x = 0 \text{ (value-based choosing)} \end{cases}$$
	`default: threshold = 1`
	`epsilon-first: threshold = 20, epsilon = NA, lambda = NA`

epsilon	[numeric] A parameter used in the **epsilon-greedy** exploration strategy. It defines the probability of making a completely random choice, as opposed to choosing based on the relative values of the left and right options. For example, if `epsilon = 0.1`, the subject has a 10 choice and a 90 relevant when `threshold` is at its default value (1) and `lambda` is not set. $$P(x) = \begin{cases} \epsilon, & x = 1 \text{ (random choosing)} \\ 1 - \epsilon, & x = 0 \text{ (value-based choosing)} \end{cases}$$ `epsilon-greedy: threshold = 1, epsilon = 0.1, lambda = NA`

`lambda` [vector] A numeric value that controls the decay rate of exploration probability in the **epsilon-decreasing** strategy. A higher `lambda` value means the probability of random choice will decrease more rapidly as the number of trials increases.

$$P(x) = \begin{cases} \frac{1}{1+\lambda \cdot trial}, & x = 1 \text{ (random choosing)} \\ \frac{\lambda \cdot trial}{1+\lambda \cdot trial}, & x = 0 \text{ (value-based choosing)} \end{cases}$$

`epsilon-decreasing` `threshold` = `1`, `epsilon` = `NA`, `lambda` = `0.5`

`alpha` [vector] Extra parameters that may be used in functions.

`beta` [vector] Extra parameters that may be used in functions.

Value

A numeric value, either 0 or 1. 0 indicates no exploration (choice based on value), and 1 indicates exploration (random choice) for that trial.

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the `if-else` statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_epsilon <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Free Parameters
  threshold = 1,
  epsilon = NA,
  lambda = NA,
  # Extra parameters
  alpha,
  beta
){
  set.seed(i)
  # Epsilon-First: random choosing before a certain trial number
  if (i <= threshold) {
```

```
  try <- 1
} else if (i > threshold & is.na(epsilon) & is.na(lambda)) {
  try <- 0
# Epsilon-Greedy: random choosing throughout the experiment with probability epsilon
} else if (i > threshold & !(is.na(epsilon)) & is.na(lambda)){
  try <- sample(
    c(1, 0),
    prob = c(epsilon, 1 - epsilon),
    size = 1
  )
# Epsilon-Decreasing: probability of random choosing decreases as trials increase
} else if (i > threshold & is.na(epsilon) & !(is.na(lambda))) {
  try <- sample(
    c(1, 0),
    prob = c(
      1 / (1 + lambda * i),
      lambda * i / (1 + lambda * i)
    ),
    size = 1
  )
}
else {
  try <- "ERROR"
}

return(try)
}

## End(Not run)
```

`func_eta` *Function: Learning Rate*

Description

The structure of `eta` depends on the model type:

- **Temporal Difference (TD) model**: `eta` is a single numeric value representing the learning rate.
- **Risk-Sensitive Temporal Difference (RSTD) model**: `eta` is a numeric vector of length two, where `eta[1]` represents the learning rate for "good" outcomes, which means the reward is higher than the expected value. `eta[2]` represents the learning rate for "bad" outcomes, which means the reward is lower than the expected value.

Usage

```
func_eta(
  i,
```

```
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  var1 = NA,
  var2 = NA,
  value,
  utility,
  reward,
  occurrence,
  eta,
  alpha,
  beta
)
```

Arguments

i	The current row number.
L_freq	The frequency of left option appearance
R_freq	The frequency of right option appearance
L_pick	The number of times left option was picked
R_pick	The number of times left option was picked
L_value	The value of the left option
R_value	The value of the right option
var1	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. default: var1 = "Extra_Var1"
var2	[character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model. default: var2 = "Extra_Var2"
value	The expected value of the stimulus in the subject's mind at this point in time.
utility	The subjective value that the subject assigns to the objective reward.
reward	The objective reward received by the subject after selecting a stimulus.
occurrence	The number of times the same stimulus has been chosen.
eta	[vector] Parameters used in the Learning Rate Function, rate_func representing the rate at which the subject updates the difference (prediction error) between the reward and the expected value in the subject's mind. The structure of eta depends on the model type:

- For the **Temporal Difference (TD) model**, where a single learning rate is used throughout the experiment

$$V_{new} = V_{old} + \eta \cdot (R - V_{old})$$

- For the **Risk-Sensitive Temporal Difference (RDTD) model**, where two different learning rates are used depending on whether the reward is lower or higher than the expected value:

$$V_{new} = V_{old} + \eta_{+} \cdot (R - V_{old}), R > V_{old}$$

$$V_{new} = V_{old} + \eta_{-} \cdot (R - V_{old}), R < V_{old}$$

`TD: eta = 0.3`

`RSTD: eta = c(0.3, 0.7)`

`alpha` [vector] Extra parameters that may be used in functions.

`beta` [vector] Extra parameters that may be used in functions.

Value

learning rate eta

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the `if-else` statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_eta <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Expected value for this stimulus
  value,
  # Subjective utility
  utility,
  # Reward observed after choice
  reward,
```

```
  # Occurrence count for this stimulus
  occurrence,

  # Free Parameter
  eta,
  # Extra parameters
  alpha,
  beta
){
################################## [ TD ] ###################################
  if (length(eta) == 1) {
    eta <- as.numeric(eta)
  }
################################ [ RSTD ] ###################################
  else if (length(eta) == 2 & utility < value) {
    eta <- eta[1]
  }
  else if (length(eta) == 2 & utility >= value) {
    eta <- eta[2]
  }
################################ [ ERROR ] ##################################
  else {
    eta <- "ERROR" # Error check
  }
  return(eta)
}

## End(Not run)
```

`func_gamma` *Function: Utility Function*

Description

This function represents an exponent used in calculating utility from reward. Its application varies depending on the specific model:

- **Stevens' Power Law**: Here, utility is calculated by raising the reward to the power of `gamma`. This describes how the subjective value (utility) of a reward changes non-linearly with its objective magnitude.
- **Kahneman's Prospect Theory**: This theory applies exponents differently for gains and losses, and introduces a loss aversion coefficient:
  - For positive rewards (gains), utility is the reward raised to the power of `gamma[1]`.
  - For negative rewards (losses), utility is calculated by first multiplying the reward by `beta`, and then raising this product to the power of `gamma[2]`. Here, `beta` acts as a loss aversion parameter, accounting for the greater psychological impact of losses compared to equivalent gains.

Usage

```
func_gamma(
  i,
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  var1 = NA,
  var2 = NA,
  value,
  utility,
  reward,
  occurrence,
  gamma = 1,
  alpha,
  beta
)
```

Arguments

i	The current row number.
L_freq	The frequency of left option appearance
R_freq	The frequency of right option appearance
L_pick	The number of times left option was picked
R_pick	The number of times left option was picked
L_value	The value of the left option
R_value	The value of the right option
var1	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. default: var1 = "Extra_Var1"
var2	[character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model. default: var2 = "Extra_Var2"
value	The expected value of the stimulus in the subject's mind at this point in time.
utility	The subjective value that the subject assigns to the objective reward.
reward	The objective reward received by the subject after selecting a stimulus.
occurrence	The number of times the same stimulus has been chosen.
gamma	[vector] This parameter represents the exponent in utility functions, fcun_gamma, specifically:

- **Stevens' Power Law**: Utility is modeled as:

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

- **Kahneman's Prospect Theory**: This exponent is applied differently based on the sign of the reward:

$$U(R) = \begin{cases} R^{\gamma_1}, & R > 0 \\ \beta \cdot R^{\gamma_2}, & R < 0 \end{cases}$$

`alpha` [vector] Extra parameters that may be used in functions.

`beta` [vector] Extra parameters that may be used in functions.

Value

Discount rate and utility

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the `if-else` statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_gamma <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Expected value for this stimulus
  value,
  # Subjective utility
  utility,
  # Reward observed after choice
  reward,
  # Occurrence count for this stimulus
  occurrence,

  # Free Parameter
  gamma = 1,
```

```
  # Extra parameters
  alpha,
  beta
){
############################## [ Utility ] ###################################
  if (length(gamma) == 1) {
    gamma <- as.numeric(gamma)
    utility <- sign(reward) * (abs(reward) ^ gamma)
  }
############################### [ Error ] ###################################
  else {
    utility <- "ERROR"
  }
  return(list(gamma, utility))
}

## End(Not run)
```

`func_logl` *Function: Loss Function*

Description

This loss function reflects the similarity between human choices and RL model predictions. If a human selects the left option and the RL model predicts a high probability for the left option, then $logP_L$ approaches 0, causing the first term to approach 0.

Since the human chose the left option, B_R becomes 0, making the second term naturally zero. Therefore, the more consistent the RL model's prediction is with human choice, the closer this LL value is to 0. Conversely, it approaches negative infinity.

$$LL = \sum B_L \times \log P_L + \sum B_R \times \log P_R$$

Usage

```
func_logl(
  i,
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  L_dir,
  R_dir,
  L_prob,
  R_prob,
  var1 = NA,
```

```
  var2 = NA,
  LR,
  try,
  value,
  utility,
  reward,
  occurrence,
  alpha,
  beta
)
```

Arguments

i	The current row number.
L_freq	The frequency of left option appearance
R_freq	The frequency of right option appearance
L_pick	The number of times left option was picked
R_pick	The number of times left option was picked
L_value	The value of the left option
R_value	The value of the right option
L_dir	Whether the participant chose the left option.
R_dir	Whether the participant chose the right option.
L_prob	The probability that the model assigns to choosing the left option.
R_prob	The probability that the model assigns to choosing the left option.
var1	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. default: var1 = "Extra_Var1"
var2	[character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model. default: var2 = "Extra_Var2"
LR	Are you calculating the probability for the left option or the right option?
try	If the choice was random, the value is 1; If the choice was based on value, the value is 0.
value	The expected value of the stimulus in the subject's mind at this point in time.
utility	The subjective value that the subject assigns to the objective reward.
reward	The objective reward received by the subject after selecting a stimulus.
occurrence	The number of times the same stimulus has been chosen.
alpha	[vector] Extra parameters that may be used in functions.
beta	[vector] Extra parameters that may be used in functions.

Value

log-likelihood

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the `if-else` statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_logl <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  #
  L_dir,
  R_dir,
  #
  L_prob,
  R_prob,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Whether calculating probability for left or right choice
  LR,
  # Is it a random choosing trial?
  try,

  # Extra parameters
  alpha,
  beta
){
  logl <- switch(
    EXPR = LR,
    "L" = L_dir * log(L_prob),
    "R" = R_dir * log(R_prob)
  )
}

## End(Not run)
```

func_pi *Function: Upper-Confidence-Bound*

Description

Unlike epsilon-greedy, which explores indiscriminately, UCB is a more intelligent exploration strategy. It biases the value of each action based on how often it has been selected. For options chosen fewer times, or those with high uncertainty, a larger "uncertainty bonus" is added to their estimated value. This increases their selection probability, effectively encouraging the exploration of potentially optimal, yet unexplored actions. A higher `pi` indicates a greater bias toward giving less-chosen options.

Usage

```
func_pi(
  i,
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  var1,
  var2,
  LR,
  pi,
  alpha,
  beta
)
```

Arguments

`i`	The current row number.
`L_freq`	The frequency of left option appearance
`R_freq`	The frequency of right option appearance
`L_pick`	The number of times left option was picked
`R_pick`	The number of times left option was picked
`L_value`	The value of the left option
`R_value`	The value of the right option
`var1`	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. `default: var1 = "Extra_Var1"`

var2 [character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model.

default: var2 = "Extra_Var2"

LR Are you calculating the probability for the left option or the right option?

pi [vector] Parameter used in the Upper-Confidence-Bound (UCB) action selection formula. func_pi controls the degree of exploration by scaling the uncertainty bonus given to less-explored options. A larger value of pi (denoted as c in Sutton and Barto(1998)) increases the influence of this bonus, leading to more exploration of actions with uncertain estimated values. Conversely, a smaller pi results in less exploration.

$$A_t = \arg\max_a \left[V_t(a) + \pi \sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

default: pi = NA

alpha [vector] Extra parameters that may be used in functions.

beta [vector] Extra parameters that may be used in functions.

Value

The probability of choosing this option

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the if-else statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_tau <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Whether calculating probability for left or right choice
  LR,
```

```
  # Free parameter
  pi = 0.1,
  # Extra parameters
  alpha,
  beta
){
############################## [ at least 1 ] ##################################
  if (is.na(x = pi)) {
    if (L_pick == 0 & R_pick == 0) {
      bias <- 0
    }
    else if (LR == "L" & L_pick == 0 & R_pick > 0) {
      bias <- 1e+4
    }
    else if (LR == "R" & R_pick == 0 & L_pick > 0) {
      bias <- 1e+4
    }
    else {
      bias <- 0
    }
  }
############################## [ bias value ] ##################################
  else if (!(is.na(x = pi)) & LR == "L") {
    bias <- pi * sqrt(log(L_pick + exp(1)) / (L_pick + 1e-10))
  }
  else if (!(is.na(x = pi)) & LR == "R") {
    bias <- pi * sqrt(log(R_pick + exp(1)) / (R_pick + 1e-10))
  }
############################## [ error ] #######################################
  else {
    bias <- "ERROR"
  }

  return(bias)
}

## End(Not run)
```

---

func_tau *Function: Soft-Max Function*

Description

The softmax function describes a probabilistic choice rule. It implies that options with higher subjective values are chosen with a greater probability, rather than deterministic. This probability of choosing the higher-valued option increases with the parameter `tau`. A higher `tau` indicates greater sensitivity to value differences, making choices more deterministic.

Usage

```
func_tau(
  i,
  L_freq,
  R_freq,
  L_pick,
  R_pick,
  L_value,
  R_value,
  var1 = NA,
  var2 = NA,
  LR,
  try,
  tau,
  lapse,
  alpha,
  beta
)
```

Arguments

i	[numeric] The current row number.
L_freq	[numeric] The frequency of left option appearance
R_freq	[numeric] The frequency of right option appearance
L_pick	[numeric] The number of times left option was picked
R_pick	[numeric] The number of times left option was picked
L_value	[numeric] The value of the left option with bias (if pi != 0)
R_value	[numeric] The value of the right option with bias (if pi != 0)
var1	[character] Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model. default: var1 = "Extra_Var1"
var2	[character] Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model. default: var2 = "Extra_Var2"

`LR` [character]

Are you calculating the probability for the left option or the right option?

`LR = "L"; LR = "R"`

`try` [numeric]

If the choice was random, the value is 1; If the choice was based on value, the value is 0.

`tau` [vector]

Parameters used in the Soft-Max Function. `prob_func` representing the sensitivity of the subject to the value difference when making decisions. It determines the probability of selecting the left option versus the right option based on their values. A larger value of tau indicates greater sensitivity to the value difference between the options. In other words, even a small difference in value will make the subject more likely to choose the higher-value option.

$$P_L = \frac{1}{1 + e^{-(V_L - V_R)\cdot\tau}}; P_R = \frac{1}{1 + e^{-(V_R - V_L)\cdot\tau}}$$

`e.g., tau = c(0.5)`

`lapse` [numeric]

A numeric value between 0 and 1, representing the lapse rate.

You can interpret this parameter as the probability of the agent "slipping" or making a random choice, irrespective of the learned action values. This accounts for moments of inattention or motor errors. In this sense, it represents the minimum probability with which any given option will be selected. It is a free parameter that acknowledges that individuals do not always make decisions with full concentration throughout an experiment.

From a modeling perspective, the lapse rate is crucial for preventing the log-likelihood calculation from returning `-Inf`. This issue arises when the model assigns a probability of zero to an action that the participant actually chose (`log(0)` is undefined). By ensuring every option has a non-zero minimum probability, the `lapse` parameter makes the fitting process more stable and robust against noise in the data.

$$P_{final} = (1 - lapse) \cdot P_{softmax} + \frac{lapse}{N_{choices}}$$

`default: lapse = 0.02`

This ensures each option has a minimum selection probability of 1 percent in TAFC tasks.

`alpha` [vector]

Extra parameters that may be used in functions.

`beta` [vector]

Extra parameters that may be used in functions.

Value

The probability of choosing this option

Note

When customizing these functions, please ensure that you do not modify the arguments. Instead, only modify the `if-else` statements or the internal logic to adapt the function to your needs.

Examples

```
## Not run:
func_tau <- function(
  # Trial number
  i,
  # Number of times this option has appeared
  L_freq,
  R_freq,
  # Number of times this option has been chosen
  L_pick,
  R_pick,
  # Current value of this option
  L_value,
  R_value,
  # Extra variables
  var1 = NA,
  var2 = NA,

  # Whether calculating probability for left or right choice
  LR,
  # Is it a random choosing trial?
  try,

  # Free parameters
  tau,
  # Extra parameters
  alpha,
  beta
){
############################### [ random ] ###################################
  if (try == 1) {
    prob <- 0.5
  }
############################### [ greedy-max ] ###############################
  else if (try == 0 & LR == "L" & is.na(tau)) {
    if (L_value == R_value) {
      prob <- 0.5
    }
    else if (L_value > R_value) {
      prob <- 1
    }
    else if (L_value < R_value) {
      prob <- 0
    }
  }
  else if (try == 0 & LR == "R" & is.na(tau)) {
    if (L_value == R_value) {
```

```
      prob <- 0.5
    }
    else if (R_value > L_value) {
      prob <- 1
    }
    else if (R_value < L_value) {
      prob <- 0
    }
  }
################################ [ soft-max ] ################################
  else if (try == 0 & LR == "L" & !(is.na(tau))) {
    prob <- 1 / (1 + exp(-(L_value - R_value) * tau))
  }
  else if (try == 0 & LR == "R" & !(is.na(tau))) {
    prob <- 1 / (1 + exp(-(R_value - L_value) * tau))
  }
################################ [ error ] ###################################
  else {
    prob <- "ERROR"
  }
################################ [ lapse ] ###################################

  prob <- (1 - lapse) * prob + (lapse / 2)

  return(prob)
}

## End(Not run)
```

`Mason_2024_G1` *Group 1 from Mason et al. (2024)*

Description

This dataset originates from Experiment 1 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 4.

- B: 90% gain 0 and 10% gain 40.
- C: 100% lose 4.
- D: 90% lose 0 and 10% lose 40.

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

- A: 100% gain 4.
- B: 90% gain 0 and 10% gain 40.
- C: 100% lose 4.
- D: 90% lose 0 and 10% lose 40.

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.

Examples

```
# Load the Mason_2024_G1 dataset
data(binaryRL::Mason_2024_G1)
head(binaryRL::Mason_2024_G1)
```

`Mason_2024_G2` *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.

Examples

```
# Load the Mason_2024_G2 dataset
data(binaryRL::Mason_2024_G2)
head(binaryRL::Mason_2024_G2)
```

`optimize_para` *Process: Optimizing Parameters*

Description

This is an internal helper function for `fit_p`. Its primary purpose is to provide a unified interface for users to interact with various optimization algorithm packages. It adapts the inputs and outputs to be compatible with eight distinct algorithms, ensuring a seamless experience regardless of the underlying solver used.

The function provides several optimization algorithms:

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

For more information, please refer to the homepage of this package: https://yuki-961004.github.io/binaryRL/

Usage

```
optimize_para(
  estimate = "MLE",
  policy = "off",
  data,
  id,
  n_trials,
  n_params,
  obj_func,
  lower,
  upper,
  priors = NULL,
  initial_params = NA,
  initial_size = 50,
  iteration = 10,
  seed = 123,
  algorithm
)
```

Arguments

`estimate` [string]

Estimation method. Can be either `"MLE"` or `"MAP"`.

- **Maximum Likelihood Estimation** `"MLE"`: (Default): This method finds the parameter values that maximize the log-likelihood of the data. A higher log-likelihood indicates that the parameters provide a better explanation for the observed human behavior. In other words, data simulated using these parameters would most closely resemble the actual human data. This method does not consider any prior information about the parameters.
- **Maximum A Posteriori Estimation** `"MAP"`: This method finds the parameter values that maximize the posterior probability. It is an iterative process based on the Expectation-Maximization (EM) framework.
  - **Initialization**: The process begins by assuming a uniform distribution as the prior for each parameter, making the initial log-prior zero. The first optimization is thus equivalent to MLE.
  - **Iteration**: After finding the best parameters for all subjects, the algorithm assesses the actual distribution of each parameter and fits a normal distribution to it. This fitted distribution becomes the new empirical prior.
  - **Re-estimation**: The parameters are then re-optimized to maximize the updated posterior probability.
  - **Convergence**: This cycle repeats until the posterior probability converges or the maximum number of iterations is reached.

  Using this method requires that the `priors` argument be specified to define the initial prior distributions.

default: `estimate = "MLE"`

`policy` [character]

Specifies the learning policy to be used. This determines how the model updates action values based on observed or simulated choices. It can be either `"off"` or `"on"`.

- **Off-Policy (Q-learning):** This is the most common approach for modeling reinforcement learning in Two-Alternative Forced Choice (TAFC) tasks. In this mode, the model's goal is to learn the underlying value of each option by observing the human participant's behavior. It achieves this by consistently updating the value of the option that the human actually chose. The focus is on understanding the value representation that likely drove the participant's decisions.
- **On-Policy (SARSA):** In this mode, the target policy and the behavior policy are identical. The model first computes the selection probability for each option based on their current values. Critically, it then uses these probabilities to sample its own action. The value update is then performed on the action that the model itself selected. This approach focuses more on directly mimicking the stochastic choice patterns of the agent, rather than just learning the underlying values from a fixed sequence of actions.

default: `policy = "off"`

`data` [data.frame]

This data should include the following mandatory columns:

- "sub"
- "time_line" (e.g., "Block", "Trial")
- "L_choice"
- "R_choice"
- "L_reward"
- "R_reward"
- "sub_choose"

`id` [character]

Specifies the ID of the subject whose optimal parameters will be fitted. This parameter accepts either string or numeric values. The provided ID must correspond to an existing subject identifier within the raw dataset provided to the function.

`n_trials` [integer]

The total number of trials in your experiment.

`n_params` [integer]

The number of free parameters in your model.

`obj_func` [function]

The objective function that the optimization algorithm package accepts. This function must strictly take only one argument, `fit_p` (a vector of model parameters). Its output must be a single numeric value representing the loss function to be minimized. For more detailed requirements and examples, please refer to the relevant documentation (`TD`, `RSTD`, `Utility`).

`lower` [vector]

Lower bounds of free parameters

`upper` [vector]

Upper bounds of free parameters

`priors` [List]

A list specifying the prior distributions for the model parameters. This argument is mandatory when using `estimate = "MAP"`. There are two primary scenarios for its use:

1. Static MAP Estimation (Non-Hierarchical) This approach is used when you have a strong, pre-defined belief about the parameter priors and do not want the model to update them iteratively.

Configuration:
- Set `estimate = "MAP"`.
- Provide `pirors` defining probability density function of free parameters
- Keep `iteration_g = 0` (the default).

Behavior: The algorithm maximizes the posterior probability based solely on your specified priors. It will not use the EM (Expectation-Maximization) framework to learn new priors from the data.

2. Hierarchical Bayesian Estimation via EM This approach is used to let the model learn the group-level (hierarchical) prior distributions directly from the data.

Configuration:
- Set `estimate = "MAP"`.
- Specify a weak or non-informative initial prior, such as a uniform distribution for all free parameters.
- Set `iteration_g` to a value greater than 0.

Behavior: With a uniform prior, the initial log-posterior equals the log-likelihood, making the first estimation step equivalent to MLE. The algorithm then initiates the EM procedure: it iteratively assesses the actual parameter distribution across all subjects and updates the group-level priors. This cycle continues until the posterior converges or `iteration_g` is reached.

default: `priors = NULL`

`initial_params` [vector]

Initial values for the free parameters that the optimization algorithm will search from. These are primarily relevant when using algorithms that require an explicit starting point, such as `L-BFGS-B`. If not specified, the function will automatically generate initial values close to zero.

default: `initial_params = NA`.

`initial_size` [integer]

This parameter corresponds to the population size in genetic algorithms (`GA`). It specifies the number of initial candidate solutions that the algorithm starts with for its evolutionary search. This parameter is only required for optimization algorithms that operate on a population, such as 'GA' or 'DEoptim'.

default: `initial_size = 50`.

`iteration` [integer]

The number of iterations the optimization algorithm will perform when searching for the best-fitting parameters during the fitting phase. A higher number of iterations may increase the likelihood of finding a global optimum but also increases computation time.

`seed` [integer]

Random seed. This ensures that the results are reproducible and remain the same each time the function is run.

default: `seed = 123`

`algorithm` [character]

Choose an algorithm package from `L-BFGS-B`, `GenSA`,`GA`,`DEoptim`,`PSO`, `Bayesian`, `CMA-ES`.

In addition, any algorithm from the `nloptr` package is also supported. If your chosen `nloptr` algorithm requires a local search, you need to input a character vector. The first element represents the algorithm used for global search, and the second element represents the algorithm used for local search.

Value

the result of binaryRL with optimal parameters

Examples

```
## Not run:
binaryRL.res <- binaryRL::optimize_para(
  data = binaryRL::Mason_2024_G2,
  id = 1,
  obj_func = binaryRL::RSTD,
  n_params = 3,
  n_trials = 360,
  lower = c(0, 0, 0),
  upper = c(1, 1, 1),
  iteration = 10,
  seed = 123,
  #algorithm = "L-BFGS-B"   # Gradient-Based (stats)
  #algorithm = "GenSA"    # Simulated Annealing (GenSA)
  #algorithm = "GA"       # Genetic Algorithm (GA)
  #algorithm = "DEoptim"  # Differential Evolution (DEoptim)
  #algorithm = "PSO"      # Particle Swarm Optimization (pso)
  #algorithm = "Bayesian" # Bayesian Optimization (mlrMBO)
  #algorithm = "CMA-ES"   # Covariance Matrix Adapting (cmaes)
  algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")
)
summary(binaryRL.res)

## End(Not run)
```

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

Description

This function is designed for model and parameter recovery of user-created (black-box) models, provided they conform to the specified interface. (demo: `TD`, `RSTD`, `Utility`). The process involves generating synthetic datasets. First, parameters are randomly sampled within a defined range. These parameters are then used to simulate artificial datasets.

Subsequently, all candidate models are used to fit these simulated datasets. Model recoverability is assessed if a synthetic dataset generated by Model A is consistently best fitted by Model A itself.

Furthermore, the function allows users to evaluate parameter recoverability. If, for instance, a synthetic dataset generated by Model A was based on parameters like 0.3 and 0.7, and Model A then recovers optimal parameters close to 0.3 and 0.7 from this data, it indicates that the parameters of Model A are recoverable.

The function provides several optimization algorithms:

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

For more information, please refer to the homepage of this package: https://yuki-961004.github.io/binaryRL/

Usage

```
rcv_d(
  estimate = "MLE",
  policy = "off",
  data,
  id = NULL,
  n_trials = NULL,
  funcs = NULL,
  model_names = c("TD", "RSTD", "Utility"),
  simulate_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  rfun = NULL,
  fit_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  dfun = NULL,
  lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
```

```
  upper = list(c(1, 5), c(1, 1, 5), c(1, 1, 5)),
  initial_params = NA,
  initial_size = 50,
  tolerance = 0.001,
  seed = 123,
  iteration_s = 100,
  iteration_f = 100,
  nc = 1,
  algorithm
)
```

Arguments

`estimate` [string]

Estimation method. Can be either `"MLE"` or `"MAP"`.

- **Maximum Likelihood Estimation** `"MLE"`: (Default): This method finds the parameter values that maximize the log-likelihood of the data. A higher log-likelihood indicates that the parameters provide a better explanation for the observed human behavior. In other words, data simulated using these parameters would most closely resemble the actual human data. This method does not consider any prior information about the parameters.
- **Maximum A Posteriori Estimation** `"MAP"`: This method finds the parameter values that maximize the posterior probability. It is an iterative process based on the Expectation-Maximization (EM) framework.
  - **Initialization**: The process begins by assuming a uniform distribution as the prior for each parameter, making the initial log-prior zero. The first optimization is thus equivalent to MLE.
  - **Iteration**: After finding the best parameters for all subjects, the algorithm assesses the actual distribution of each parameter and fits a normal distribution to it. This fitted distribution becomes the new empirical prior.
  - **Re-estimation**: The parameters are then re-optimized to maximize the updated posterior probability.
  - **Convergence**: This cycle repeats until the posterior probability converges or the maximum number of iterations is reached.

  Using this method requires that the `priors` argument be specified to define the initial prior distributions.

default: `estimate = "MLE"`

`policy` [string]

Specifies the learning policy to be used. This determines how the model updates action values based on observed or simulated choices. It can be either `"off"` or `"on"`.

- **Off-Policy (Q-learning):** This is the most common approach for modeling reinforcement learning in Two-Alternative Forced Choice (TAFC) tasks. In this mode, the model's goal is to learn the underlying value of each option by observing the human participant's behavior. It achieves this by consistently updating the value of the option that the human actually chose. The

focus is on understanding the value representation that likely drove the participant's decisions.

- **Off-Policy (SARSA):** In this mode, the target policy and the behavior policy are identical. The model first computes the selection probability for each option based on their current values. Critically, it then uses these probabilities to sample its own action. The value update is then performed on the action that the model itself selected. This approach focuses more on directly mimicking the stochastic choice patterns of the agent, rather than just learning the underlying values from a fixed sequence of actions.

default: `policy = "off"`

`data` [data.frame]

This data should include the following mandatory columns:

- `sub` "Subject"
- `time_line` "Block" "Trial"
- `L_choice` "L_choice"
- `R_choice` "R_choice"
- `L_reward` "L_reward"
- `R_reward` "R_reward"
- `sub_choose` "Sub_Choose"

`id` [CharacterVector]

Specifies which subject's data to use. In parameter and model recovery analyses, the specific subject ID is often irrelevant. Although the experimental trial order might have some randomness for each subject, the sequence of reward feedback is typically pseudo-random.

The default value for this argument is `NULL`. When `id = NULL`, the program automatically detects existing subject IDs within the dataset. It then randomly selects one subject as a sample, and the parameter and model recovery procedures are performed based on this selected subject's data.

default: `id = NULL`

`n_trials` [integer]

Represents the total number of trials a single subject experienced in the experiment. If this parameter is kept at its default value of `NULL`, the program will automatically detect how many trials a subject experienced from the provided data. This information is primarily used for calculating model fit statistics such as AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion).

default: `n_trials = NULL`

`funcs` [CharacterVector]

A character vector containing the names of all user-defined functions required for the computation. When parallel computation is enabled (i.e., `nc > 1`), user-defined models and their custom functions might not be automatically accessible within the parallel environment.

Therefore, if you have created your own reinforcement learning model that modifies the package's default six default functions (default functions: `util_func =`

func_gamma, rate_func = func_eta, expl_func = func_epsilon, bias_func = func_pi, prob_func = func_tau, loss_func = func_logl), you must explicitly provide the names of your custom functions as a vector here.

model_names [List]

The names of fit modals

e.g. model_names = c("TD", "RSTD", "Utility")

simulate_models

[List]

A list of functions used to simulate data from different models. Each function in the list should represent one model.

e.g. simulate_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility)

rfun [List]

A nested list of functions used to generate random parameter values for simulation. The top-level elements of the list should be named according to the models. Each of these elements must be a named list of functions, where each name corresponds to a model parameter and its value is the random number generation function.

e.g., stats::runif, stats::rexp

fit_models [List]

A list of functions used to fit different models to the data. Each function in the list should represent one model.

e.g. fit_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility)

dfun [List]

A nested list that defines the probability density/mass functions (PDF/PMF) for each model's parameters. The top-level names of the list must match the model names. Each element must be another named list, where each name corresponds to a model parameter and its value is the probability density function.

e.g., stats::dunif, stats::dexp

lower [List]

The lower bounds of models' free parameters.

e.g. lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0))

upper [List]

The upper bounds of models' free parameters.

e.g. upper = list(c(1, 1), c(1, 1, 1), c(1, 1, 10))

initial_params [NumericVector]

Initial values for the free parameters that the optimization algorithm will search from. These are primarily relevant when using algorithms that require an explicit starting point, such as L-BFGS-B. If not specified, the function will automatically generate initial values close to zero.

default: initial_params = NA.

initial_size [integer]

This parameter corresponds to the population size in genetic algorithms (GA). It specifies the number of initial candidate solutions that the algorithm starts with for its evolutionary search. This parameter is only required for optimization algorithms that operate on a population, such as GA or DEoptim.

default: initial_size = 50.

tolerance [double]

Convergence threshold for MAP estimation. If the change in log posterior probability between iterations is smaller than this value, the algorithm is considered to have converged and the program will stop.

default: tolerance = 0.001

seed [integer]

Random seed. This ensures that the results are reproducible and remain the same each time the function is run.

default: seed = 123

iteration_s [integer]

This parameter determines how many simulated datasets are created for subsequent model and parameter recovery analyses.

default: iteration_s = 10

iteration_f [NumericVector]

The number of iterations for the optimization algorithm. The required format depends on the estimation method used.

- If estimate = "MLE", this should be a single numeric value specifying the total number of iterations.
- If estimate = "MAP", this should be a NumericVector of length two: c(MLE_iterations, MAP_iterations). (e.g. iteration_f = c(100, 10))

A higher number of iterations may increase the likelihood of finding a global optimum but also increases computation time.

default: iteration_f = 10

nc [integer]

Number of cores to use for parallel processing. Since fitting optimal parameters for each subject is an independent task, parallel computation can significantly speed up the fitting process:

- **nc = 1**: The fitting proceeds sequentially. Parameters for one subject are fitted completely before moving to the next subject.
- **nc > 1**: The fitting is performed in parallel across subjects. For example, if nc = 4, the algorithm will simultaneously fit data for four subjects. Once these are complete, it will proceed to fit the next batch of subjects (e.g., subjects 5-8), and so on, until all subjects are processed.

default: nc = 1

algorithm [string]

Choose an algorithm package from L-BFGS-B, GenSA,GA,DEoptim,PSO, Bayesian, CMA-ES.

In addition, any algorithm from the nloptr package is also supported. If your chosen nloptr algorithm requires a local search, you need to input a character vector. The first element represents the algorithm used for global search, and the second element represents the algorithm used for local search.
e.g. algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")

Value

A list where each element is a data.frame. Each data.frame within this list records the results of fitting synthetic data (generated by Model A) with Model B.

Note

While both fit_p and rcv_d utilize the same underlying optimize_para function to find optimal parameters, they play distinct and sequential roles in the modeling pipeline.

The key differences are as follows:

1. **Purpose and Data Source**: rcv_d should always be performed before fit_p. Its primary role is to validate a model's stability by fitting it to synthetic data generated by the model itself. This process, known as parameter recovery, ensures the model is well-behaved. In contrast, fit_p is used in the subsequent stage to fit the validated model to real experimental data.
2. **Estimation Method**: rcv_d does not include an estimate argument. This is because the synthetic data is generated from known "true" parameters, which are drawn from pre-defined distributions (typically uniform for most parameters and exponential for the inverse temperature). Since the ground truth is known, a hierarchical estimation (MAP) is not applicable. The fit_p function, however, requires this argument to handle real data where the true parameters are unknown.
3. **Policy Setting**: In fit_p, the policy setting has different effects: "on-policy" is better for learning choice patterns, while "off-policy" yields more accurate parameter estimates. For rcv_d, the process defaults to an "off-policy" approach because its main objectives are to verify if the true parameters can be accurately recovered and to assess whether competing models are distinguishable, tasks for which off-policy estimation is more suitable.

Examples

```
## Not run:
recovery <- binaryRL::rcv_d(
  data = binaryRL::Mason_2024_G2,
#+-----------------------------------------------------------------------------+#
#|---------------------------- black-box function ----------------------------|#
  funcs = c("your_funcs"),
  estimate = c("MLE", "MAP"),
  policy = c("off", "on"),
  model_names = c("TD", "RSTD", "Utility"),
#|------------------------------ simulate models ------------------------------|#
  simulate_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  rfun = list(
    list(
      eta = function() { stats::runif(n = 1, min = 0, max = 1) },
      tau = function() { stats::rexp(n = 1, rate = 1) }
```

```
    ),
    list(
      etan = function() { stats::runif(n = 1, min = 0, max = 1) },
      etap = function() { stats::runif(n = 1, min = 0, max = 1) },
      tau = function() { stats::rexp(n = 1, rate = 1) }
    ),
    list(
      eta = function() { stats::runif(n = 1, min = 0, max = 1) },
      gamma = function() { stats::runif(n = 1, min = 0, max = 1) },
      tau = function() { stats::rexp(n = 1, rate = 1) }
    ),
  ),
#|---------------------------------- fit models -------------------------------|#
  fit_models = list(binaryRL::TD, binaryRL::RSTD, binaryRL::Utility),
  dfun = list(
    list(
      eta = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
      tau = function(x) { stats::dexp(x, rate = 1, log = TRUE) }
    ),
    list(
      etan = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
      etap = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
      tau = function(x) { stats::dexp(x, rate = 1, log = TRUE) }
    ),
    list(
      eta = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
      gamma = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
      tau = function(x) { stats::dexp(x, rate = 1, log = TRUE) }
    ),
  ),
#|---------------------------------- bound ------------------------------------|#
  lower = list(c(0, 0), c(0, 0, 0), c(0, 0, 0)),
  upper = list(c(1, 5), c(1, 1, 5), c(1, 1, 5)),
#|---------------------------- interation number ----------------------------|#
  iteration_s = 100,
  iteration_f = c(100, 10),
#|-------------------------------- algorithms ---------------------------------|#
  nc = 1,                 # <nc > 1>: parallel computation across subjects
  # Base R Optimization
  #algorithm = "L-BFGS-B"  # Gradient-Based (stats)
#|-----------------------------------------------------------------------------|#
  # Specialized External Optimization
  #algorithm = "GenSA"    # Simulated Annealing (GenSA)
  #algorithm = "GA"       # Genetic Algorithm (GA)
  #algorithm = "DEoptim"  # Differential Evolution (DEoptim)
  #algorithm = "PSO"      # Particle Swarm Optimization (pso)
  #algorithm = "Bayesian" # Bayesian Optimization (mlrMBO)
  #algorithm = "CMA-ES"   # Covariance Matrix Adapting (cmaes)
#|-----------------------------------------------------------------------------|#
  # Optimization Library (nloptr)
  algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")
#|-------------------------------- algorithms ---------------------------------|#
#+#############################################################################+#
```

```
)

result <- dplyr::bind_rows(recovery) %>%
  dplyr::select(simulate_model, fit_model, iteration, everything())

# Ensure the output directory exists
if (!dir.exists("../OUTPUT")) {
  dir.create("../OUTPUT", recursive = TRUE)
}

write.csv(result, file = "../OUTPUT/result_recovery.csv", row.names = FALSE)

## End(Not run)
```

`recovery_data` *Process: Recovering Fake Data*

Description

This function processes the synthetic datasets generated by `simulate_list()`. For each of these simulated datasets, it then fits every model specified within the `fit_model` list. In essence, it iteratively calls the `optimize_para()` function for each generated object.

The fitting procedure is analogous to that performed by `fit_p`, and it similarly leverages parallel computation across subjects to significantly accelerate the parameter estimation process.

Usage

```
recovery_data(
  policy,
  estimate,
  list,
  id = 1,
  n_trials,
  n_params,
  funcs = NULL,
  model_name,
  fit_model,
  dfun,
  lower,
  upper,
  initial_params = NA,
  initial_size = 50,
  tolerance,
  seed = 123,
  iteration,
  nc = 1,
  algorithm
)
```

Arguments

`policy` [character]

Specifies the learning policy to be used. This determines how the model updates action values based on observed or simulated choices. It can be either `"off"` or `"on"`.

- **Off-Policy (Q-learning):** This is the most common approach for modeling reinforcement learning in Two-Alternative Forced Choice (TAFC) tasks. In this mode, the model's goal is to learn the underlying value of each option by observing the human participant's behavior. It achieves this by consistently updating the value of the option that the human actually chose. The focus is on understanding the value representation that likely drove the participant's decisions.
- **On-Policy (SARSA):** In this mode, the target policy and the behavior policy are identical. The model first computes the selection probability for each option based on their current values. Critically, it then uses these probabilities to sample its own action. The value update is then performed on the action that the model itself selected. This approach focuses more on directly mimicking the stochastic choice patterns of the agent, rather than just learning the underlying values from a fixed sequence of actions.

default: `policy = "off"`

`estimate` [string]

Estimation method. Can be either `"MLE"` or `"MAP"`.

- **Maximum Likelihood Estimation** `"MLE"`: (Default): This method finds the parameter values that maximize the log-likelihood of the data. A higher log-likelihood indicates that the parameters provide a better explanation for the observed human behavior. In other words, data simulated using these parameters would most closely resemble the actual human data. This method does not consider any prior information about the parameters.
- **Maximum A Posteriori Estimation** `"MAP"`: This method finds the parameter values that maximize the posterior probability. It is an iterative process based on the Expectation-Maximization (EM) framework.
  - **Initialization**: The process begins by assuming a uniform distribution as the prior for each parameter, making the initial log-prior zero. The first optimization is thus equivalent to MLE.
  - **Iteration**: After finding the best parameters for all subjects, the algorithm assesses the actual distribution of each parameter and fits a normal distribution to it. This fitted distribution becomes the new empirical prior.
  - **Re-estimation**: The parameters are then re-optimized to maximize the updated posterior probability.
  - **Convergence**: This cycle repeats until the posterior probability converges or the maximum number of iterations is reached.

  Using this method requires that the `priors` argument be specified to define the initial prior distributions.

default: `estimate = "MLE"`

`list` [list]

A list generated by function `simulate_list()`

`id` [vector]

Specifies which subject's data to use. In parameter and model recovery analyses, the specific subject ID is often irrelevant. Although the experimental trial order might have some randomness for each subject, the sequence of reward feedback is typically pseudo-random.

The default value for this argument is `NULL`. When `id = NULL`, the program automatically detects existing subject IDs within the dataset. It then randomly selects one subject as a sample, and the parameter and model recovery procedures are performed based on this selected subject's data.

default: `id = NULL`

`n_trials` [integer]

The total number of trials in your experiment.

`n_params` [integer]

The number of free parameters in your model.

`funcs` [character]

A character vector containing the names of all user-defined functions required for the computation. When parallel computation is enabled (i.e., 'nc > 1'), user-defined models and their custom functions might not be automatically accessible within the parallel environment.

Therefore, if you have created your own reinforcement learning model that modifies the package's default four default functions (default functions: `util_func = func_gamma`, `rate_func = func_eta`, `expl_func = func_epsilon` `bias_func = func_pi` `prob_func = func_tau`), you must explicitly provide the names of your custom functions as a vector here.

`model_name` [character]

The name of modal

`fit_model` [function]

fit model object function

`dfun` [List]

A nested list that defines the probability density/mass functions (PDF/PMF) for each model's parameters. The top-level names of the list must match the model names. Each element must be another named list, where each name corresponds to a model parameter and its value is the probability density function.

e.g., `stats::dunif`, `stats::dexp`

`lower` [List]

The lower bounds of model's free parameters.

e.g. `lower = c(0, 0, 0)`

`upper` [List]

The upper bounds of model's free parameters.

e.g. `upper = c(1, 1, 5)`

`initial_params` [numeric]

Initial values for the free parameters that the optimization algorithm will search from. These are primarily relevant when using algorithms that require an explicit starting point, such as `L-BFGS-B`. If not specified, the function will automatically generate initial values close to zero.

default: `initial_params = NA`.

`initial_size` [integer]

This parameter corresponds to the population size in genetic algorithms (`GA`). It specifies the number of initial candidate solutions that the algorithm starts with for its evolutionary search. This parameter is only required for optimization algorithms that operate on a population, such as 'GA' or 'DEoptim'.

default: `initial_size = 50`.

`tolerance` [double]

Convergence threshold for MAP estimation. If the change in log posterior probability between iterations is smaller than this value, the algorithm is considered to have converged and the program will stop.

default: `tolerance = 0.001`

`seed` [integer]

Random seed. This ensures that the results are reproducible and remain the same each time the function is run.

default: `seed = 123`

`iteration` [integer]

The number of iterations the optimization algorithm will perform when searching for the best-fitting parameters during the fitting phase. A higher number of iterations may increase the likelihood of finding a global optimum but also increases computation time.

`nc` [integer]

Number of cores to use for parallel processing. Since fitting optimal parameters for each subject is an independent task, parallel computation can significantly speed up the fitting process:

- **'nc = 1'**: The fitting proceeds sequentially. Parameters for one subject are fitted completely before moving to the next subject.
- **'nc > 1'**: The fitting is performed in parallel across subjects. For example, if 'nc = 4', the algorithm will simultaneously fit data for four subjects. Once these are complete, it will proceed to fit the next batch of subjects (e.g., subjects 5-8), and so on, until all subjects are processed.

default: `nc = 1`

`algorithm` [character] Choose an algorithm package from `L-BFGS-B`, `GenSA`,`GA`,`DEoptim`,`PSO`, `Bayesian`, `CMA-ES`.

In addition, any algorithm from the `nloptr` package is also supported. If your chosen `nloptr` algorithm requires a local search, you need to input a character vector. The first element represents the algorithm used for global search, and the second element represents the algorithm used for local search.

Value

a data frame for parameter recovery and model recovery

Examples

```
## Not run:
df_recovery <- recovery_data(
  list = list_simulated,
  policy = "off",
  estimate = "MAP",
  model_name = "RSTD",
  fit_model = binaryRL::RSTD,
  dfun = list(
    etan = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
    etap = function(x) { stats::dunif(x, min = 0, max = 1, log = TRUE) },
    tau = function(x) { stats::dexp(x, rate = 1, log = TRUE) }
  ),
  lower = c(0, 0, 0),
  upper = c(1, 1, 10),
  iteration = c(10, 3),
  nc = 10,
  algorithm = c("NLOPT_GN_MLSL", "NLOPT_LN_BOBYQA")
)

## End(Not run)
```

---

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

Description

After completing Step 3 using `fit_p()` to obtain the optimal parameters for each subject and saving the resulting CSV locally, this function allows you to load that result dataset. It then applies these optimal parameters back into the reinforcement learning model, effectively simulating how the "robot" (the model) would make its choices.

Based on this generated dataset, you can then analyze the robot's data in the same manner as you would analyze human behavioral data. If a particular model's fitted data can successfully reproduce the experimental effects observed in human subjects, it strongly suggests that this model is a good and valid representation of the process.

Usage

```
rpl_e(
  data,
  id = NULL,
  result,
  model,
  model_name,
```

```
  param_prefix = "param_",
  n_trials = NULL
)
```

Arguments

`data`	[data.frame] This data should include the following mandatory columns: • `sub` "Subject" • `time_line` "Block" "Trial" • `L_choice` "L_choice" • `R_choice` "R_choice" • `L_reward` "L_reward" • `R_reward` "R_reward" • `sub_choose` "Sub_Choose"
`id`	[CharacterVector] A vector specifying the subject ID(s) for which parameters should be fitted. The function will process only the subjects provided in this vector. To fit all subjects, you can either explicitly set the argument as `id = unique(data$Subject)` or leave it as the default (`id = NULL`). Both approaches will direct the function to fit parameters for every unique subject in the dataset. It is strongly recommended to avoid using simple numeric sequences like `id = 1:4`. This practice can lead to errors if subject IDs are stored as strings (e.g., subject four is stored as "004") or are not sequentially numbered. default: `id = NULL`
`result`	[data.frame] Output data generated by the `fit_p()` function. Each row represents model fit results for a subject.
`model`	[Function] A model function to be applied in evaluating the experimental effect.
`model_name`	[string] A character string specifying the name of the model to extract from the result.
`param_prefix`	[string] A prefix string used to identify parameter columns in the result data default: `param_prefix = "param_"`
`n_trials`	[integer] Represents the total number of trials a single subject experienced in the experiment. If this parameter is kept at its default value of `NULL`, the program will automatically detect how many trials a subject experienced from the provided data. This information is primarily used for calculating model fit statistics such as AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion). default: `n_trials = NULL`

Value

A list, where each element is a data.frame representing one subject's results. Each data.frame includes the value update history for each option, the learning rate (`eta`), utility function (`gamma`), and other relevant information used in each update.

Examples

```
## Not run:
list <- list()

list[[1]] <- dplyr::bind_rows(
  binaryRL::rpl_e(
    data = binaryRL::Mason_2024_G2,
    result = read.csv("../OUTPUT/result_comparison.csv"),
    model = binaryRL::TD,
    model_name = "TD"
  )
)

list[[2]] <- dplyr::bind_rows(
  binaryRL::rpl_e(
    data = binaryRL::Mason_2024_G2,
    result = read.csv("../OUTPUT/result_comparison.csv"),
    model = binaryRL::RSTD,
    model_name = "RSTD"
  )
)

list[[3]] <- dplyr::bind_rows(
  binaryRL::rpl_e(
    data = binaryRL::Mason_2024_G2,
    result = read.csv("../OUTPUT/result_comparison.csv"),
    model = binaryRL::Utility,
    model_name = "Utility"
  )
)

## End(Not run)
```

`RSTD` *Model: RSTD*

Description

$$V_{new} = V_{old} + \eta_{+} \cdot (R - V_{old}), R > V_{old}$$
$$V_{new} = V_{old} + \eta_{-} \cdot (R - V_{old}), R < V_{old}$$

Usage

```
RSTD(params)
```

Arguments

params	[vector] algorithm packages accept only one argument

Value

loss [numeric]

algorithm packages accept only one return

Examples

```
## Not run:
RSTD <- function(params) {
  res <- binaryRL::run_m(
    data = data,
    id = id,
    eta = c(params[1], params[2]),
    tau = c(params[3]),
    priors = priors,
    n_params = n_params,
    n_trials = n_trials,
    mode = mode,
    policy = policy
  )

  assign(x = "binaryRL.res", value = res, envir = binaryRL.env)
  loss <- switch(EXPR = estimate, "MLE" = -res$ll, "MAP" = -res$lpo)
  switch(EXPR = mode, "fit" = loss, "simulate" = res, "replay" = res)
}

## End(Not run)
```

---

run_m *Step 1: Building reinforcement learning model*

Description

This function is designed to construct and customize reinforcement learning models.

Items for model construction:

- **Data Input and Specification:** You must provide the raw dataset for analysis. Crucially, you need to inform the run_m function about the corresponding column names within your dataset (e.g., Mason_2024_G1, Mason_2024_G2). This is a game, so it's critical that your dataset includes rewards for both the human-chosen option and the unchosen options.

- **Customizable RL Models:** This function allows you to define and adjust the number of free parameters to create various reinforcement learning models.
  - *Value Function:*
    - *Learning Rate:* By adjusting the number of `eta`, you can construct basic reinforcement learning models such as Temporal Difference (TD) and Risk Sensitive Temporal Difference (RSTD). You can also directly adjust `func_eta` to define your own custom learning rate function.
    - *Utility Function:* You can directly adjust the form of `func_gamma` to incorporate the principles of Kahneman's Prospect Theory. Currently, the built-in `func_gamma` only takes the form of a power function, consistent with Stevens' Power Law.
  - *Exploration-Exploitation Trade-off:*
    - *Initial Values:* This involves setting the initial expected value for each option when it hasn't been chosen yet. A higher initial value encourages exploration.
    - *Epsilon:* Adjusting the `threshold`, `epsilon` and `lambda` parameters can lead to exploration strategies such as epsilon-first, epsilon-greedy, or epsilon-decreasing.
    - *Upper-Confidence-Bound:* By adjusting `pi`, it controls the degree of exploration by scaling the uncertainty bonus given to less-explored options.
    - *Soft-Max:* By adjusting the inverse temperature parameter `tau`, this controls the agent's sensitivity to value differences. A higher value of tau means greater emphasis on value differences, leading to more exploitation. A smaller value of tau indicates a greater tendency towards exploration.
- **Objective Function Format for Optimization:** Once your model is defined in `run_m`, it must be structured as an objective function that accepts `params` as input and returns a loss value (typically `logL`). This format ensures compatibility with the **algorithm** package, which uses it to estimate optimal parameters. For an example of a standard objective function format, see `TD`, `RSTD`, `Utility`.

For more information, please refer to the homepage of this package: https://yuki-961004.github.io/binaryRL/

Usage

```
run_m(
  name = NA,
  mode = c("simulate", "fit", "replay"),
  policy = c("on", "off"),
  data,
  id,
  n_params,
  n_trials,
  gamma = 1,
  eta,
  initial_value = NA_real_,
  threshold = 1,
  epsilon = NA,
  lambda = NA,
  pi = NA,
```

```
  tau = NA,
  lapse = 0.02,
  alpha = NA,
  beta = NA,
  priors = NULL,
  util_func = func_gamma,
  rate_func = func_eta,
  expl_func = func_epsilon,
  bias_func = func_pi,
  prob_func = func_tau,
  loss_func = func_logl,
  sub = "Subject",
  time_line = c("Block", "Trial"),
  L_choice = "L_choice",
  R_choice = "R_choice",
  L_reward = "L_reward",
  R_reward = "R_reward",
  sub_choose = "Sub_Choose",
  rob_choose = "Rob_Choose",
  raw_cols = NULL,
  var1 = NA_character_,
  var2 = NA_character_,
  seed = 123,
  digits_1 = 2,
  digits_2 = 5,
  engine = "cpp"
)
```

Arguments

`name`	[string] The name of your RL model
`mode`	[string] This parameter controls the function's operational mode. It has three possible values, each typically associated with a specific function: • `"simulate"`: Should be used when working with `rcv_d`. • `"fit"`: Should be used when working with `fit_p`. • `"replay"`: Should be used when working with `rpl_e`. In most cases, you won't need to modify this parameter directly, as suitable default values are set for different contexts.
`policy`	[string] Specifies the learning policy to be used. This determines how the model updates action values based on observed or simulated choices. It can be either `"off"` or `"on"`. • **Off-Policy (Q-learning):** This is the most common approach for modeling reinforcement learning in Two-Alternative Forced Choice (TAFC) tasks. In

this mode, the model's goal is to learn the underlying value of each option by observing the human participant's behavior. It achieves this by consistently updating the value of the option that the human actually chose. The focus is on understanding the value representation that likely drove the participant's decisions.

- **On-Policy (SARSA):** In this mode, the target policy and the behavior policy are identical. The model first computes the selection probability for each option based on their current values. Critically, it then uses these probabilities to sample its own action. The value update is then performed on the action that the model itself selected. This approach focuses more on directly mimicking the stochastic choice patterns of the agent, rather than just learning the underlying values from a fixed sequence of actions.

`data` [data.frame]

This data should include the following mandatory columns:

- `sub` "Subject"
- `time_line` "Block" "Trial"
- `L_choice` "L_choice"
- `R_choice` "R_choice"
- `L_reward` "L_reward"
- `R_reward` "R_reward"
- `sub_choose` "Sub_Choose"

`id` [string]

Which subject is going to be analyzed. The value should correspond to an entry in the "sub" column, which must contain the subject IDs.

e.g. `id = 18`

`n_params` [integer]

The number of free parameters in your model.

`n_trials` [integer]

The total number of trials in your experiment.

`gamma` [NumericVector]

Note: This should not be confused with the discount rate parameter (also named gamma) found in Temporal Difference (TD) models. Rescorla-Wagner model does not include a discount rate. Here, `gamma` is used as a free parameter to shape the utility function.

- **Stevens' Power Law**: Utility is modeled as:

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

- **Kahneman's Prospect Theory**: This exponent is applied differently based on the sign of the reward:

$$U(R) = \begin{cases} R^{\gamma_1}, & R > 0 \\ \beta \cdot R^{\gamma_2}, & R < 0 \end{cases}$$

default: `gamma = 1`

`eta` [NumericVector]

Parameters used in the Learning Rate Function, `rate_func`, representing the rate at which the subject updates the difference (prediction error) between the reward and the expected value in the subject's mind.

The structure of `eta` depends on the model type:

- For the **Temporal Difference (TD) model**, where a single learning rate is used throughout the experiment

$$V_{new} = V_{old} + \eta \cdot (R - V_{old})$$

- For the **Risk-Sensitive Temporal Difference (RDTD) model**, where two different learning rates are used depending on whether the reward is lower or higher than the expected value:

$$V_{new} = V_{old} + \eta_{+} \cdot (R - V_{old}), R > V_{old}$$

$$V_{new} = V_{old} + \eta_{-} \cdot (R - V_{old}), R < V_{old}$$

TD: `eta = 0.3`

RSTD: `eta = c(0.3, 0.7)`

`initial_value` [double]

Subject's initial expected value for each stimulus's reward. If this value is not set `initial_value = NA`, the subject will use the reward received after the first trial as the initial value for that stimulus. In other words, the learning rate for the first trial is 100

default: `initial_value = NA_real_`

`threshold` [integer]

Controls the initial exploration phase in the **epsilon-first** strategy. This is the number of early trials where the subject makes purely random choices, as they haven't yet learned the options' values. For example, `threshold = 20` means random choices for the first 20 trials. For **epsilon-greedy** or **epsilon-decreasing** strategies, `threshold` should be kept at its default value.

$$P(x) = \begin{cases} \text{trial} \leq \text{threshold}, & x = 1 \text{ (random choosing)} \\ \text{trial} > \text{threshold}, & x = 0 \text{ (value-based choosing)} \end{cases}$$

default: `threshold = 1`

epsilon-first: `threshold = 20, epsilon = NA, lambda = NA`

`epsilon` [NumericVector]

A parameter used in the **epsilon-greedy** exploration strategy. It defines the probability of making a completely random choice, as opposed to choosing based on the relative values of the left and right options. For example, if `epsilon = 0.1`, the subject has a 10 choice and a 90 relevant when `threshold` is at its default value (1) and `lambda` is not set.

$$P(x) = \begin{cases} \epsilon, & x = 1 \text{ (random choosing)} \\ 1 - \epsilon, & x = 0 \text{ (value-based choosing)} \end{cases}$$

epsilon-greedy: `threshold = 1, epsilon = 0.1, lambda = NA`

`lambda` [NumericVector]

A numeric value that controls the decay rate of exploration probability in the **epsilon-decreasing** strategy. A higher `lambda` value means the probability of random choice will decrease more rapidly as the number of trials increases.

$$P(x) = \begin{cases} \frac{1}{1+\lambda \cdot trial}, & x = 1 \text{ (random choosing)} \\ \frac{\lambda \cdot trial}{1+\lambda \cdot trial}, & x = 0 \text{ (value-based choosing)} \end{cases}$$

epsilon-decreasing: `threshold = 1, epsilon = NA, lambda = 0.5`

`pi` [NumericVector]

Parameter used in the Upper-Confidence-Bound (UCB) action selection formula. `bias_func` controls the degree of exploration by scaling the uncertainty bonus given to less-explored options. A larger value of `pi` (denoted as `c` in Sutton and Barto(2018)) increases the influence of this bonus, leading to more exploration of actions with uncertain estimated values. Conversely, a smaller `pi` results in less exploration.

$$A_t = \arg\max_a \left[V_t(a) + \pi \sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

default: `pi = NA`

`tau` [NumericVector]

Parameters used in the Soft-Max Function. `prob_func` representing the sensitivity of the subject to the value difference when making decisions. It determines the probability of selecting the left option versus the right option based on their values. A larger value of tau indicates greater sensitivity to the value difference between the options. In other words, even a small difference in value will make the subject more likely to choose the higher-value option.

$$P_L = \frac{1}{1 + e^{-(V_L - V_R) \cdot \tau}}; P_R = \frac{1}{1 + e^{-(V_R - V_L) \cdot \tau}}$$

default `tau = NA`

`lapse` [double]

A numeric value between 0 and 1, representing the lapse rate.

You can interpret this parameter as the probability of the agent "slipping" or making a random choice, irrespective of the learned action values. This accounts for moments of inattention or motor errors. In this sense, it represents the minimum probability with which any given option will be selected. It is a free parameter that acknowledges that individuals do not always make decisions with full concentration throughout an experiment.

From a modeling perspective, the lapse rate is crucial for preventing the log-likelihood calculation from returning `-Inf`. This issue arises when the model assigns a probability of zero to an action that the participant actually chose (`log(0)` is undefined). By ensuring every option has a non-zero minimum probability, the `lapse` parameter makes the fitting process more stable and robust against noise in the data.

$$P_{final} = (1 - lapse) \cdot P_{softmax} + \frac{lapse}{N_{choices}}$$

default: `lapse = 0.02`

This ensures each option has a minimum selection probability of 1 percent in TAFC tasks.

`alpha` [NumericVector]

Extra parameters that may be used in functions.

`beta` [NumericVector]

Extra parameters that may be used in functions.

`priors` [list]

A list specifying the prior distributions for the model parameters. This argument is mandatory when using `estimate = "MAP"`.

default: `priors = NULL`

`util_func` [Function]

Utility Function see `func_gamma`.

`rate_func` [Function]

Learning Rate Function see `func_eta`.

`expl_func` [Function]

Exploration Strategy Function see `func_epsilon`.

`bias_func` [Function]

Upper-Confidence-Bound see `func_pi`.

`prob_func` [Function]

Soft-Max Function see `func_tau`.

`loss_func` [Function]

Loss Function see `func_logl`.

`sub` [string]

Column name of subject ID

e.g. `sub = "Subject"`

`time_line` [CharacterVector]

A vector specifying the name of the column that the sequence of the experiment. This argument defines how the experiment is structured, such as whether it is organized by "Block" with breaks in between, and multiple trials within each block.

default: `time_line = c("Block", "Trial")`

`L_choice` [string]

Column name of left choice.

default: `L_choice = "Left_Choice"`

`R_choice` [string]

Column name of right choice.

default: `R_choice = "Right_Choice"`

L_reward [string]
Column name of the reward of left choice
default: L_reward = "Left_reward"

R_reward [string]
Column name of the reward of right choice
default: R_reward = "Right_reward"

sub_choose [string]
Column name of choices made by the subject.
default: sub_choose = "Choose"

rob_choose [string]
Column name of choices made by the model, which you could ignore.
default: rob_choose = "Rob_Choose"

raw_cols [CharacterVector]
Defaults to NULL. If left as NULL, it will directly capture all column names from the raw data.

var1 [string]
Column name of extra variable 1. If your model uses more than just reward and expected value, and you need other information, such as whether the choice frame is Gain or Loss, then you can input the 'Frame' column as var1 into the model.
default: var1 = NA_character_

var2 [string]
Column name of extra variable 2. If one additional variable, var1, does not meet your needs, you can add another additional variable, var2, into your model.
default: var2 = NA_character_

seed [integer]
Random seed. This ensures that the results are reproducible and remain the same each time the function is run.
default: seed = 123

digits_1 [integer]
The number of decimal places to retain for columns related to value function
default: digits_1 = 2

digits_2 [integer]
The number of decimal places to retain for columns related to select function.
default: digits_2 = 5

engine [string]
- "r": Use the pure R version of the code.
- "cpp": Use the Rcpp-optimized version.
default: engine = "cpp"

Value

A list of class binaryRL containing the results of the model fitting.

Examples

```
data <- binaryRL::Mason_2024_G2

binaryRL.res <- binaryRL::run_m(
  mode = "replay",
  data = data,
  id = 18,
  eta = c(0.321, 0.765),
  tau = 0.5,
  n_params = 3,
  n_trials = 360
)

summary(binaryRL.res)
```

`simulate_list` *Process: Simulating Fake Data*

Description

This function generates random input parameters for a model based on user-specified distributions. For example, if the first parameter, `eta`, is set to follow a uniform distribution from 0 to 1, its values will be randomly sampled from `U(0, 1)`.

You can also specify parameters to be drawn from a normal distribution. For example, `eta = function() { stats::rnorm(n = 1, mean = 0.5, sd = 0.1) }`. Make sure the last parameter, which typically represents the inverse temperature parameter in the soft-max function, is sampled from an exponential distribution.

Usage

```
simulate_list(
  data,
  id = 1,
  n_params,
  n_trials,
  obj_func,
  rfun,
  iteration = 10,
  seed = 123
)
```

Arguments

`data` [data.frame]

This data should include the following mandatory columns:

- "sub"

- "time_line" (e.g., "Block", "Trial")
- "L_choice"
- "R_choice"
- "L_reward"
- "R_reward"
- "sub_choose"

`id` [vector]

Specifies which subject's data to use. In parameter and model recovery analyses, the specific subject ID is often irrelevant. Although the experimental trial order might have some randomness for each subject, the sequence of reward feedback is typically pseudo-random.

The default value for this argument is `NULL`. When `id = NULL`, the program automatically detects existing subject IDs within the dataset. It then randomly selects one subject as a sample, and the parameter and model recovery procedures are performed based on this selected subject's data.

default: `id = NULL`

`n_params` [integer]

The number of free parameters in your model.

`n_trials` [integer]

The total number of trials in your experiment.

`obj_func` [function]

The objective function that the optimization algorithm package accepts. This function must strictly take only one argument, `params` (a vector of model parameters). Its output must be a single numeric value representing the loss function to be minimized. For more detailed requirements and examples, please refer to the relevant documentation (`TD`, `RSTD`, `Utility`).

`rfun` [List]

A nested list of functions used to generate random parameter values for simulation. The top-level elements of the list should be named according to the models. Each of these elements must be a named list of functions, where each name corresponds to a model parameter and its value is the random number generation function.

e.g., `stats::runif`, `stats::rexp`

`iteration` [integer]

This parameter determines how many simulated datasets are created for subsequent model and parameter recovery analyses.

default: `iteration_s = 10`

`seed` [integer]

Random seed. This ensures that the results are reproducible and remain the same each time the function is run.

default: `seed = 123`

Value

a list with fake data generated by random free parameters

Examples

```
## Not run:
list_simulated <- binaryRL::simulate_list(
  data = binaryRL::Mason_2024_G2,
  obj_func = binaryRL::RSTD,
  n_params = 3,
  n_trials = 360,
  rfun = list(
    etan = function() { stats::runif(n = 1, min = 0, max = 1) },
    etap = function() { stats::runif(n = 1, min = 0, max = 1) },
    tau = function() { stats::rexp(n = 1, rate = 1) }
  ),
  iteration = 10
)

## End(Not run)
```

`summary.binaryRL` *S3method summary*

Description

S3method summary

Usage

```
## S3 method for class 'binaryRL'
summary(object, ...)
```

Arguments

`object`	binaryRL result
`...`	others

Value

summary

TD *Model: TD*

Description

$$V_{new} = V_{old} + \eta \cdot (R - V_{old})$$

Usage

```
TD(params)
```

Arguments

params	[vector] algorithm packages accept only one argument

Value

loss [numeric]

algorithm packages accept only one return

Examples

```
## Not run:
TD <- function(params) {
  res <- binaryRL::run_m(
    data = data,
    id = id,
    eta = c(params[1]),
    tau = c(params[2]),
    priors = priors,
    n_params = n_params,
    n_trials = n_trials,
    mode = mode,
    policy = policy
  )

  assign(x = "binaryRL.res", value = res, envir = binaryRL.env)
  loss <- switch(EXPR = estimate, "MLE" = -res$ll, "MAP" = -res$lpo)
  switch(EXPR = mode, "fit" = loss, "simulate" = res, "replay" = res)
}

## End(Not run)
```

`Utility` *Model: Utility*

Description

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

$$V_{new} = V_{old} + \eta \cdot (U(R) - V_{old})$$

Usage

```
Utility(params)
```

Arguments

`params`	[vector] algorithm packages accept only one argument

Value

loss [numeric]

algorithm packages accept only one return

Examples

```
## Not run:
Utility <- function(params) {
  res <- binaryRL::run_m(
    data = data,
    id = id,
    eta = c(params[1]),
    gamma = c(params[2]),
    tau = c(params[3]),
    priors = priors,
    n_params = n_params,
    n_trials = n_trials,
    mode = mode,
    policy = policy
  )

  assign(x = "binaryRL.res", value = res, envir = binaryRL.env)
  loss <- switch(EXPR = estimate, "MLE" = -res$ll, "MAP" = -res$lpo)
  switch(EXPR = mode, "fit" = loss, "simulate" = res, "replay" = res)
}

## End(Not run)
```

Index

fit_p, 2, *50*
func_epsilon, *5*, 9, *37*, *43*, *54*
func_eta, *5*, 12, *37*, *43*, *49*, *54*
func_gamma, *5*, 15, *37*, *43*, *49*, *54*
func_logl, *5*, 18, *37*, *54*
func_pi, *5*, 21, *37*, *43*, *54*
func_tau, *5*, 23, *37*, *43*, *54*

Mason_2024_G1, 27, *48*
Mason_2024_G2, 28, *48*

optimize_para, 29

rcv_d, 34, *50*
recovery_data, 41
rpl_e, 45, *50*
RSTD, *2*, *31*, *34*, 47, *49*, *57*
run_m, 48, *48, 49*

simulate_list, 56
summary.binaryRL, 58

TD, *2*, *31*, *34*, *49*, *57*, 59

Utility, *2*, *31*, *34*, *49*, *57*, 60