Package ‘BayesMallows’

November 4, 2022

Type Package

Title Bayesian Preference Learning with the Mallows Rank Model

Version 1.2.1

Maintainer Oystein Sorensen <oystein.sorensen.1985@gmail.com>

Description An implementation of the Bayesian version of the Mallows rank model
(Vitelli et al., Journal of Machine Learning Research, 2018 <https://jmlr.org/papers/v18/15-481.html>;
Crispino et al., Annals of Applied Statistics, 2019 <doi:10.1214/18-AOAS1203>). Both Metropolis-Hastings
and sequential Monte Carlo algorithms for estimating the models are available. Cayley, footrule, Hamming, Kendall, Spearman, and Ulam distances are supported in the models. The rank data to be
analyzed can be in the form of complete rankings, top-k rankings, partially missing rankings, as well
as consistent and inconsistent pairwise preferences. Several functions for plotting and studying the posterior distributions of parameters are provided. The package also provides functions for estimating
the partition function (normalizing constant) of the Mallows rank model, both with the importance sampling algorithm of Vitelli et al. and asymptotic approximation with the IPFP algorithm (Mukherjee, Annals of Statistics, 2016 <doi:10.1214/15-AOS1389>).

URL https://github.com/ocbe-uio/BayesMallows

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.2.1

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.0), ggplot2 (>= 3.1.0), Rdpack (>= 1.0), igraph (>= 1.2.5), sets (>= 1.0-18), relations (>= 0.6-8), rlang (>= 0.3.1), HDInterval (>= 0.2.0), cowplot (>= 1.0.0)

LinkingTo Rcpp, RcppArmadillo, testthat

Suggests R.rsp, xml2, knitr, testthat (>= 3.0.0), label.switching (>= 1.7), rmarkdown, covr, parallel (>= 3.5.1)

VignetteBuilder R.rsp, knitr

RdMacros Rdpack

NeedsCompilation yes

Author Oystein Sorensen [aut, cre] (<https://orcid.org/0000-0003-0724-3542>),
Valeria Vitelli [aut] (<https://orcid.org/0000-0002-6746-0453>),
Marta Crispino [aut],
Qinghua Liu [aut],
Cristina Mollica [aut],
Luca Tardella [aut],
Anja Stein [aut],
Waldir Leoncio [ctr]

Repository CRAN

Date/Publication 2022-11-04 15:10:02 UTC

R topics documented:

`assess_convergence` *Trace Plots from Metropolis-Hastings Algorithm*

Description

`assess_convergence` provides trace plots for the parameters of the Mallows Rank model, in order to study the convergence of the Metropolis-Hastings algorithm.

Usage

```
assess_convergence(
  model_fit,
  parameter = "alpha",
  items = NULL,
  assessors = NULL
)
```

Arguments

`model_fit`	A fitted model object of class `BayesMallows` returned from `compute_mallows` or an object of class `BayesMallowsMixtures` returned from `compute_mallows_mixtures`.
`parameter`	Character string specifying which parameter to plot. Available options are `"alpha"`, `"rho"`, `"Rtilde"`, `"cluster_probs"`, or `"theta"`.

`items` The items to study in the diagnostic plot for `rho`. Either a vector of item names, corresponding to `model_fit$items` or a vector of indices. If NULL, five items are selected randomly. Only used when `parameter = "rho"` or `parameter = "Rtilde"`.

`assessors` Numeric vector specifying the assessors to study in the diagnostic plot for `"Rtilde"`.

See Also

`compute_mallows`, `plot.BayesMallows`

`assign_cluster` *Assign Assessors to Clusters*

Description

Assign assessors to clusters by finding the cluster with highest posterior probability.

Usage

```
assign_cluster(
  model_fit,
  burnin = model_fit$burnin,
  soft = TRUE,
  expand = FALSE
)
```

Arguments

`model_fit` An object of type `BayesMallows`, returned from `compute_mallows`.

`burnin` A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.

`soft` A logical specifying whether to perform soft or hard clustering. If `soft=TRUE`, all cluster probabilities are returned, whereas if `soft=FALSE`, only the maximum a posterior (MAP) cluster probability is returned, per assessor. In the case of a tie between two or more cluster assignments, a random cluster is taken as MAP estimate.

`expand` A logical specifying whether or not to expand the rowset of each assessor to also include clusters for which the assessor has 0 a posterior assignment probability. Only used when `soft = TRUE`. Defaults to `FALSE`.

Value

A dataframe. If `soft = FALSE`, it has one row per assessor, and columns `assessor`, `probability` and `map_cluster`. If `soft = TRUE`, it has `n_cluster` rows per assessor, and the additional column `cluster`.

See Also

compute_mallows for an example where this function is used.

BayesMallows *BayesMallows: Bayesian Preference Learning with the Mallows Rank Model.*

Description

The BayesMallows package provides functionality for fully Bayesian analysis of preference or rank data. The package implements the Bayesian Mallows model described in Vitelli et al. (2018), which handles complete rankings, top-k rankings, ranks missing at random, and consistent pairwise preference data, as well as mixtures of rank models. Modeling of pairwise preferences containing inconsistencies, as described in Crispino et al. (2019), is also supported. See also Sørensen et al. (2020) for an overview of the methods and a tutorial.

The documentation and examples for the following functions are likely most useful to get you started:

- For analysis of rank or preference data, see compute_mallows.
- For computation of multiple models with varying numbers of mixture components, see compute_mallows_mixtures.
- For estimation of the partition function (normalizing constant) using either the importance sampling algorithm of Vitelli et al. (2018) or the asymptotic algorithm of Mukherjee (2016), see estimate_partition_function.

References

Crispino M, Arjas E, Vitelli V, Barrett N, Frigessi A (2019). “A Bayesian Mallows approach to nontransitive pair comparison data: How human are sounds?” *The Annals of Applied Statistics*, **13**(1), 492–519. doi:10.1214/18aoas1203.

Mukherjee S (2016). “Estimation in exponential families on permutations.” *The Annals of Statistics*, **44**(2), 853–875. doi:10.1214/15aos1389.

Sørensen Ø, Crispino M, Liu Q, Vitelli V (2020). “BayesMallows: An R Package for the Bayesian Mallows Model.” *The R Journal*, **12**(1), 324–342. doi:10.32614/RJ2020026.

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). “Probabilistic Preference Learning with the Mallows Rank Model.” *Journal of Machine Learning Research*, **18**(1), 1–49. https://jmlr.org/papers/v18/15-481.html.

beach_preferences *Beach Preferences*

Description

Example dataset from (Vitelli et al. 2018), Section 6.2.

Usage

```
beach_preferences
```

Format

An object of class `data.frame` with 1442 rows and 3 columns.

References

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). "Probabilistic Preference Learning with the Mallows Rank Model." *Journal of Machine Learning Research*, **18**(1), 1–49. `https://jmlr.org/papers/v18/15-481.html`.

calculate_backward_probability *Calculate Backward Probability*

Description

Function to calculate probability of assigning a set of specific ranks to an specific item given its rank in the consensus ranking

Usage

```
calculate_backward_probability(
  item_ordering,
  partial_ranking,
  current_ranking,
  remaining_set,
  rho,
  alpha,
  n_items,
  metric
)
```

Arguments

`item_ordering` A vector of integer values to represent the specified queue of which unranked item to assign a rank for the proposed augmented ranking

`partial_ranking` An incomplete rank sequence vector of the original observed incomplete ranking which contains NAs

`current_ranking` An complete rank sequence vector of the proposed augmented ranking obtained from calculate_forward_probability function

`remaining_set` A vector of integer values to represent the elements (ranks) missing from original observed ranking

`rho` Numeric vector specifying the consensus ranking

`alpha` Numeric value of the scale parameter

`n_items` Integer is the number of items in a ranking

`metric` A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.

Value

backward_auxiliary_ranking_probability A numerical value of creating the previous augmented ranking using the same item ordering used to create the new augmented ranking in calculate_forward_probability function.

`calculate_forward_probability` *Calculate Forward Probability*

Description

Function to calculate probability of assigning a set of specific ranks to an specific item given its rank in the consensus ranking

Usage

```
calculate_forward_probability(
  item_ordering,
  partial_ranking,
  remaining_set,
  rho,
  alpha,
  n_items,
  metric
)
```

Arguments

`item_ordering` A vector of integer values to represent the specified queue of which unranked item to assign a rank for the proposed augmented ranking

`partial_ranking` An incomplete rank sequence vector of the original observed incomplete ranking which contains NAs

`remaining_set` A vector of integer values to represent the elements (ranks) missing from original observed ranking

`rho` Numeric vector specifying the consensus ranking

`alpha` Numeric value of the scale parameter

`n_items` Integer is the number of items in a ranking

`metric` A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.

Value

List containing aug_ranking, a ranking sequence vector of the proposed augmented ranking and forward_prob a numerical value of the probability of creating the augmented ranking using the pseudolikelihood augmentation.

`compute_consensus` *Compute Consensus Ranking*

Description

Compute the consensus ranking using either cumulative probability (CP) or maximum a posteriori (MAP) consensus (Vitelli et al. 2018). For mixture models, the consensus is given for each mixture. Consensus of augmented ranks can also be computed for each assessor, by setting `parameter = "Rtilde"`.

Usage

```
compute_consensus(model_fit, ...)
```

Arguments

`model_fit` An object returned from `compute_mallows`.

`...` other arguments passed to methods.

References

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). “Probabilistic Preference Learning with the Mallows Rank Model.” *Journal of Machine Learning Research*, **18**(1), 1–49. `https://jmlr.org/papers/v18/15-481.html`.

Examples

```
# The example datasets potato_visual and potato_weighing contain complete
# rankings of 20 items, by 12 assessors. We first analyse these using the Mallows
# model:
model_fit <- compute_mallows(potato_visual)

# Se the documentation to compute_mallows for how to assess the convergence of the algorithm
# Having chosen burin = 1000, we compute posterior intervals
model_fit$burnin <- 1000
# We then compute the CP consensus.
compute_consensus(model_fit, type = "CP")
# And we compute the MAP consensus
compute_consensus(model_fit, type = "MAP")

## Not run:
  # CLUSTERWISE CONSENSUS
  # We can run a mixture of Mallows models, using the n_clusters argument
  # We use the sushi example data. See the documentation of compute_mallows for a more elaborate
  # example
  model_fit <- compute_mallows(sushi_rankings, n_clusters = 5)
  # Keeping the burnin at 1000, we can compute the consensus ranking per cluster
  model_fit$burnin <- 1000
  cp_consensus_df <- compute_consensus(model_fit, type = "CP")
  # We can now make a table which shows the ranking in each cluster:
  cp_consensus_df$cumprob <- NULL
  stats::reshape(cp_consensus_df, direction = "wide", idvar = "ranking",
                 timevar = "cluster",
                 varying = list(sort(unique(cp_consensus_df$cluster))))

## End(Not run)

## Not run:
  # MAP CONSENSUS FOR PAIRWISE PREFENCE DATA
  # We use the example dataset with beach preferences.
  model_fit <- compute_mallows(preferences = beach_preferences)
  # We set burnin = 1000
  model_fit$burnin <- 1000
  # We now compute the MAP consensus
  map_consensus_df <- compute_consensus(model_fit, type = "MAP")

## End(Not run)

## Not run:
  # CP CONSENSUS FOR AUGMENTED RANKINGS
  # We use the example dataset with beach preferences.
  model_fit <- compute_mallows(preferences = beach_preferences, save_aug = TRUE,
                               aug_thinning = 2, seed = 123L)
  # We set burnin = 1000
  model_fit$burnin <- 1000
  # We now compute the CP consensus of augmented ranks for assessors 1 and 3
  cp_consensus_df <- compute_consensus(model_fit, type = "CP",
                                       parameter = "Rtilde", assessors = c(1L, 3L))
  # We can also compute the MAP consensus for assessor 2
  map_consensus_df <- compute_consensus(model_fit, type = "MAP",
                                        parameter = "Rtilde", assessors = 2L)

  # Caution!
  # With very sparse data or with too few iterations, there may be ties in the MAP consensus
  # This is illustrated below for the case of only 5 post-burnin iterations. Two MAP rankings are
  # equally likely in this case (and for this seed).
  model_fit <- compute_mallows(preferences = beach_preferences, nmc = 1005,
                               save_aug = TRUE, aug_thinning = 1, seed = 123L)
  model_fit$burnin <- 1000
  compute_consensus(model_fit, type = "MAP", parameter = "Rtilde", assessors = 2L)

## End(Not run)
```

compute_consensus.BayesMallows

Compute Consensus Ranking

Description

Compute Consensus Ranking

Usage

```
## S3 method for class 'BayesMallows'
compute_consensus(
  model_fit,
  type = "CP",
  burnin = model_fit$burnin,
  parameter = "rho",
  assessors = 1L,
  ...
)
```

Arguments

model_fit	An object returned from compute_mallows.
type	Character string specifying which consensus to compute. Either "CP" or "MAP". Defaults to "CP".
burnin	A numeric value specifying the number of iterations to discard as burn-in. Defaults to model_fit$burnin, and must be provided if model_fit$burnin does not exist. See assess_convergence.
parameter	Character string defining the parameter for which to compute the consensus. Defaults to "rho". Available options are "rho" and "Rtilde", with the latter giving consensus rankings for augmented ranks.

`assessors` When `parameter = "rho"`, this integer vector is used to define the assessors for which to compute the augmented ranking. Defaults to `1L`, which yields augmented rankings for assessor 1.

`...` other arguments passed to methods.

`compute_consensus.consensus_SMCMallows` *Compute Consensus Ranking*

Description

Compute the consensus ranking using either cumulative probability (CP) or maximum a posteriori (MAP) consensus (Vitelli et al. 2018). For mixture models, the consensus is given for each mixture.

Usage

```
## S3 method for class 'consensus_SMCMallows'
compute_consensus(model_fit, type, burnin)
```

Arguments

`model_fit` An object returned from `compute_mallows`.

`type` Character string specifying which consensus to compute. Either `"CP"` or `"MAP"`. Defaults to `"CP"`.

`burnin` A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.

Author(s)

Anja Stein

`compute_mallows` *Preference Learning with the Mallows Rank Model*

Description

Compute the posterior distributions of the parameters of the Bayesian Mallows Rank Model, given rankings or preferences stated by a set of assessors.

The `BayesMallows` package uses the following parametrization of the Mallows rank model (Mallows 1957):

$$p(r|\alpha,\rho) = (1/Z_n(\alpha)) \exp -\alpha/n d(r,\rho)$$

where r is a ranking, α is a scale parameter, ρ is the latent consensus ranking, $Z_n(\alpha)$ is the partition function (normalizing constant), and $d(r,\rho)$ is a distance function measuring the distance between

r and ρ. Note that some authors use a Mallows model without division by n in the exponent; this includes the PerMallows package, whose scale parameter θ corresponds to α/n in the BayesMallows package. We refer to (Vitelli et al. 2018) for further details of the Bayesian Mallows model.

compute_mallows always returns posterior distributions of the latent consensus ranking ρ and the scale parameter α. Several distance measures are supported, and the preferences can take the form of complete or incomplete rankings, as well as pairwise preferences. compute_mallows can also compute mixtures of Mallows models, for clustering of assessors with similar preferences.

Usage

```
compute_mallows(
  rankings = NULL,
  preferences = NULL,
  obs_freq = NULL,
  metric = "footrule",
  error_model = NULL,
  n_clusters = 1L,
  save_clus = FALSE,
  clus_thin = 1L,
  nmc = 2000L,
  leap_size = max(1L, floor(n_items/5)),
  swap_leap = 1L,
  rho_init = NULL,
  rho_thinning = 1L,
  alpha_prop_sd = 0.1,
  alpha_init = 1,
  alpha_jump = 1L,
  lambda = 0.001,
  alpha_max = 1e+06,
  psi = 10L,
  include_wcd = (n_clusters > 1),
  save_aug = FALSE,
  aug_thinning = 1L,
  logz_estimate = NULL,
  verbose = FALSE,
  validate_rankings = TRUE,
  na_action = "augment",
  constraints = NULL,
  save_ind_clus = FALSE,
  seed = NULL
)
```

Arguments

rankings A matrix of ranked items, of size n_assessors x n_items. See create_ranking if you have an ordered set of items that need to be converted to rankings. If preferences is provided, rankings is an optional initial value of the rankings, generated by generate_initial_ranking. If rankings has column names, these are assumed to be the names of the items. NA values in rankings are treated as missing data and automatically augmented; to change this behavior, see the na_action argument.

preferences A dataframe with pairwise comparisons, with 3 columns, named assessor, bottom_item, and top_item, and one row for each stated preference. Given a set of pairwise preferences, generate a transitive closure using generate_transitive_closure. This will give preferences the class "BayesMallowsTC". If preferences is not of class "BayesMallowsTC", compute_mallows will call generate_transitive_closure on preferences before computations are done. In the current version, the pairwise preferences are assumed to be mutually compatible.

obs_freq A vector of observation frequencies (weights) to apply do each row in rankings. This can speed up computation if a large number of assessors share the same rank pattern. Defaults to NULL, which means that each row of rankings is multiplied by 1. If provided, obs_freq must have the same number of elements as there are rows in rankings, and rankings cannot be NULL. See obs_freq for more information and rank_freq_distr for a convenience function for computing it.

metric A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are "footrule", "spearman", "cayley", "hamming", "kendall", and "ulam". The distance given by metric is also used to compute within-cluster distances, when include_wcd = TRUE.

error_model Character string specifying which model to use for inconsistent rankings. Defaults to NULL, which means that inconsistent rankings are not allowed. At the moment, the only available other option is "bernoulli", which means that the Bernoulli error model is used. See Crispino et al. (2019) for a definition of the Bernoulli model.

n_clusters Integer specifying the number of clusters, i.e., the number of mixture components to use. Defaults to 1L, which means no clustering is performed. See compute_mallows_mixtures for a convenience function for computing several models with varying numbers of mixtures.

save_clus Logical specifying whether or not to save cluster assignments. Defaults to FALSE.

clus_thin Integer specifying the thinning to be applied to cluster assignments and cluster probabilities. Defaults to 1L.

nmc Integer specifying the number of iteration of the Metropolis-Hastings algorithm to run. Defaults to 2000L. See assess_convergence for tools to check convergence of the Markov chain.

leap_size Integer specifying the step size of the leap-and-shift proposal distribution. Defaults floor(n_items / 5).

swap_leap Integer specifying the step size of the Swap proposal. Only used when error_model is not NULL.

rho_init Numeric vector specifying the initial value of the latent consensus ranking ρ. Defaults to NULL, which means that the initial value is set randomly. If rho_init is provided when n_clusters > 1, each mixture component ρ_c gets the same initial value.

`rho_thinning` Integer specifying the thinning of `rho` to be performed in the Metropolis- Hastings algorithm. Defaults to `1L`. `compute_mallows` save every `rho_thinningth` value of ρ.

`alpha_prop_sd` Numeric value specifying the standard deviation of the lognormal proposal distribution used for α in the Metropolis-Hastings algorithm. Defaults to `0.1`.

`alpha_init` Numeric value specifying the initial value of the scale parameter α. Defaults to `1`. When `n_clusters > 1`, each mixture component α_c gets the same initial value.

`alpha_jump` Integer specifying how many times to sample ρ between each sampling of α. In other words, how many times to jump over α while sampling ρ, and possibly other parameters like augmented ranks $\tilde{R}$ or cluster assignments z. Setting `alpha_jump` to a high number can speed up computation time, by reducing the number of times the partition function for the Mallows model needs to be computed. Defaults to `1L`.

`lambda` Strictly positive numeric value specifying the rate parameter of the truncated exponential prior distribution of α. Defaults to `0.1`. When `n_cluster > 1`, each mixture component α_c has the same prior distribution.

`alpha_max` Maximum value of `alpha` in the truncated exponential prior distribution.

`psi` Integer specifying the concentration parameter ψ of the Dirichlet prior distribution used for the cluster probabilities $\tau_1, \tau_2, \ldots, \tau_C$, where C is the value of `n_clusters`. Defaults to `10L`. When `n_clusters = 1`, this argument is not used.

`include_wcd` Logical indicating whether to store the within-cluster distances computed during the Metropolis-Hastings algorithm. Defaults to `TRUE` if `n_clusters > 1` and otherwise `FALSE`. Setting `include_wcd = TRUE` is useful when deciding the number of mixture components to include, and is required by `plot_elbow`.

`save_aug` Logical specifying whether or not to save the augmented rankings every `aug_thinningth` iteration, for the case of missing data or pairwise preferences. Defaults to `FALSE`. Saving augmented data is useful for predicting the rankings each assessor would give to the items not yet ranked, and is required by `plot_top_k`.

`aug_thinning` Integer specifying the thinning for saving augmented data. Only used when `save_aug = TRUE`. Defaults to `1L`.

`logz_estimate` Estimate of the partition function, computed with `estimate_partition_function`. Be aware that when using an estimated partition function when `n_clusters > 1`, the partition function should be estimated over the whole range of α values covered by the prior distribution for α with high probability. In the case that a cluster α_c becomes empty during the Metropolis-Hastings algorithm, the posterior of α_c equals its prior. For example, if the rate parameter of the exponential prior equals, say $\lambda = 0.001$, there is about 37 % (or exactly: `1 -pexp(1000, 0.001)`) prior probability that $\alpha_c > 1000$. Hence when `n_clusters > 1`, the estimated partition function should cover this range, or λ should be increased.

`verbose` Logical specifying whether to print out the progress of the Metropolis-Hastings algorithm. If `TRUE`, a notification is printed every 1000th iteration. Defaults to `FALSE`.

`validate_rankings`
Logical specifying whether the rankings provided (or generated from `preferences`) should be validated. Defaults to `TRUE`. Turning off this check will reduce computing time with a large number of items or assessors.

na_action Character specifying how to deal with NA values in the rankings matrix, if provided. Defaults to "augment", which means that missing values are automatically filled in using the Bayesian data augmentation scheme described in Vitelli et al. (2018). The other options for this argument are "fail", which means that an error message is printed and the algorithm stops if there are NAs in rankings, and "omit" which simply deletes rows with NAs in them.

constraints Optional constraint set returned from generate_constraints. Defaults to NULL, which means the the constraint set is computed internally. In repeated calls to compute_mallows, with very large datasets, computing the constraint set may be time consuming. In this case it can be beneficial to precompute it and provide it as a separate argument.

save_ind_clus Whether or not to save the individual cluster probabilities in each step. This results in csv files cluster_probs1.csv, cluster_probs2.csv, ..., being saved in the calling directory. This option may slow down the code considerably, but is necessary for detecting label switching using Stephen's algorithm. See label_switching for more information.

seed Optional integer to be used as random number seed.

Value

A list of class BayesMallows.

References

Crispino M, Arjas E, Vitelli V, Barrett N, Frigessi A (2019). "A Bayesian Mallows approach to nontransitive pair comparison data: How human are sounds?" *The Annals of Applied Statistics*, **13**(1), 492–519. doi:10.1214/18aoas1203.

Mallows CL (1957). "Non-Null Ranking Models. I." *Biometrika*, **44**(1/2), 114–130.

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). "Probabilistic Preference Learning with the Mallows Rank Model." *Journal of Machine Learning Research*, **18**(1), 1–49. https://jmlr.org/papers/v18/15-481.html.

See Also

compute_mallows_mixtures for a function that computes separate Mallows models for varying numbers of clusters.

Examples

```
# ANALYSIS OF COMPLETE RANKINGS
# The example datasets potato_visual and potato_weighing contain complete
# rankings of 20 items, by 12 assessors. We first analyse these using the Mallows
# model:
model_fit <- compute_mallows(potato_visual)

# We study the trace plot of the parameters
assess_convergence(model_fit, parameter = "alpha")
```

```
## Not run: assess_convergence(model_fit, parameter = "rho")

# Based on these plots, we set burnin = 1000.
model_fit$burnin <- 1000
# Next, we use the generic plot function to study the posterior distributions
# of alpha and rho
plot(model_fit, parameter = "alpha")
## Not run: plot(model_fit, parameter = "rho", items = 10:15)

# We can also compute the CP consensus posterior ranking
compute_consensus(model_fit, type = "CP")

# And we can compute the posterior intervals:
# First we compute the interval for alpha
compute_posterior_intervals(model_fit, parameter = "alpha")
# Then we compute the interval for all the items
## Not run: compute_posterior_intervals(model_fit, parameter = "rho")

# ANALYSIS OF PAIRWISE PREFERENCES
## Not run:
  # The example dataset beach_preferences contains pairwise
  # preferences between beaches stated by 60 assessors. There
  # is a total of 15 beaches in the dataset.
  # In order to use it, we first generate all the orderings
  # implied by the pairwise preferences.
  beach_tc <- generate_transitive_closure(beach_preferences)
  # We also generate an inital rankings
  beach_rankings <- generate_initial_ranking(beach_tc, n_items = 15)
  # We then run the Bayesian Mallows rank model
  # We save the augmented data for diagnostics purposes.
  model_fit <- compute_mallows(rankings = beach_rankings,
                               preferences = beach_tc,
                               save_aug = TRUE,
                               verbose = TRUE)
  # We can assess the convergence of the scale parameter
  assess_convergence(model_fit)
  # We can assess the convergence of latent rankings. Here we
  # show beaches 1-5.
  assess_convergence(model_fit, parameter = "rho", items = 1:5)
  # We can also look at the convergence of the augmented rankings for
  # each assessor.
  assess_convergence(model_fit, parameter = "Rtilde",
                     items = c(2, 4), assessors = c(1, 2))
  # Notice how, for assessor 1, the lines cross each other, while
  # beach 2 consistently has a higher rank value (lower preference) for
  # assessor 2. We can see why by looking at the implied orderings in
  # beach_tc
  subset(beach_tc, assessor %in% c(1, 2) &
           bottom_item %in% c(2, 4) & top_item %in% c(2, 4))
  # Assessor 1 has no implied ordering between beach 2 and beach 4,
  # while assessor 2 has the implied ordering that beach 4 is preferred
  # to beach 2. This is reflected in the trace plots.
```

```
## End(Not run)

# CLUSTERING OF ASSESSORS WITH SIMILAR PREFERENCES
## Not run:
  # The example dataset sushi_rankings contains 5000 complete
  # rankings of 10 types of sushi
  # We start with computing a 3-cluster solution, and save
  # cluster assignments by setting save_clus = TRUE
  model_fit <- compute_mallows(sushi_rankings, n_clusters = 3,
                               nmc = 10000, save_clus = TRUE, verbose = TRUE)
  # We then assess convergence of the scale parameter alpha
  assess_convergence(model_fit)
  # Next, we assess convergence of the cluster probabilities
  assess_convergence(model_fit, parameter = "cluster_probs")
  # Based on this, we set burnin = 1000
  # We now plot the posterior density of the scale parameters alpha in
  # each mixture:
  model_fit$burnin <- 1000
  plot(model_fit, parameter = "alpha")
  # We can also compute the posterior density of the cluster probabilities
  plot(model_fit, parameter = "cluster_probs")
  # We can also plot the posterior cluster assignment. In this case,
  # the assessors are sorted according to their maximum a posteriori cluster estimate.
  plot(model_fit, parameter = "cluster_assignment")
  # We can also assign each assessor to a cluster
  cluster_assignments <- assign_cluster(model_fit, soft = FALSE)

## End(Not run)

# DETERMINING THE NUMBER OF CLUSTERS
## Not run:
  # Continuing with the sushi data, we can determine the number of cluster
  # Let us look at any number of clusters from 1 to 10
  # We use the convenience function compute_mallows_mixtures
  n_clusters <- seq(from = 1, to = 10)
  models <- compute_mallows_mixtures(n_clusters = n_clusters, rankings = sushi_rankings,
                                     nmc = 6000, alpha_jump = 10, include_wcd = TRUE)
  # models is a list in which each element is an object of class BayesMallows,
  # returned from compute_mallows
  # We can create an elbow plot
  plot_elbow(models, burnin = 1000)
  # We then select the number of cluster at a point where this plot has
  # an "elbow", e.g., at 6 clusters.

## End(Not run)

# SPEEDING UP COMPUTION WITH OBSERVATION FREQUENCIES
# With a large number of assessors taking on a relatively low number of unique rankings,
# the obs_freq argument allows providing a rankings matrix with the unique set of rankings,
# and the obs_freq vector giving the number of assessors with each ranking.
# This is illustrated here for the potato_visual dataset
#
# assume each row of potato_visual corresponds to between 1 and 5 assessors, as
# given by the obs_freq vector
set.seed(1234)
obs_freq <- sample.int(n = 5, size = nrow(potato_visual), replace = TRUE)
m <- compute_mallows(rankings = potato_visual, obs_freq = obs_freq)

# See the separate help page for more examples, with the following code
help("obs_freq")
```

---

compute_mallows_mixtures

Compute Mixtures of Mallows Models

Description

Convenience function for computing Mallows models with varying numbers of mixtures. This is useful for deciding the number of mixtures to use in the final model.

Usage

```
compute_mallows_mixtures(n_clusters, ..., cl = NULL)
```

Arguments

`n_clusters`	Integer vector specifying the number of clusters to use.
`...`	Other named arguments, passed to `compute_mallows`.
`cl`	Optional computing cluster used for parallelization, returned from `parallel::makeCluster`. Defaults to `NULL`.

Value

A list of Mallows models of class `BayesMallowsMixtures`, with one element for each number of mixtures that was computed. This object can be studied with `plot_elbow`.

Examples

```
# DETERMINING THE NUMBER OF CLUSTERS IN THE SUSHI EXAMPLE DATA
## Not run:
  # Let us look at any number of clusters from 1 to 10
  # We use the convenience function compute_mallows_mixtures
  n_clusters <- seq(from = 1, to = 10)
  models <- compute_mallows_mixtures(n_clusters = n_clusters,
                                     rankings = sushi_rankings,
                                     include_wcd = TRUE)
  # models is a list in which each element is an object of class BayesMallows,
  # returned from compute_mallows
  # We can create an elbow plot
  plot_elbow(models, burnin = 1000)
  # We then select the number of cluster at a point where this plot has
  # an "elbow", e.g., n_clusters = 5.
  # Having chosen the number of clusters, we can now study the final model
  # Rerun with 5 clusters, now setting save_clus = TRUE to get cluster assignments
  mixture_model <- compute_mallows(rankings = sushi_rankings, n_clusters = 5,
                                   include_wcd = TRUE, save_clus = TRUE)
  # Delete the models object to free some memory
  rm(models)
  # Set the burnin
  mixture_model$burnin <- 1000
  # Plot the posterior distributions of alpha per cluster
  plot(mixture_model)
  # Compute the posterior interval of alpha per cluster
  compute_posterior_intervals(mixture_model, parameter = "alpha")
  # Plot the posterior distributions of cluster probabilities
  plot(mixture_model, parameter = "cluster_probs")
  # Plot the posterior probability of cluster assignment
  plot(mixture_model, parameter = "cluster_assignment")
  # Plot the posterior distribution of "tuna roll" in each cluster
  plot(mixture_model, parameter = "rho", items = "tuna roll")
  # Compute the cluster-wise CP consensus, and show one column per cluster
  cp <- compute_consensus(mixture_model, type = "CP")
  cp$cumprob <- NULL
  stats::reshape(cp, direction = "wide", idvar = "ranking",
                 timevar = "cluster", varying = list(as.character(unique(cp$cluster))))

  # Compute the MAP consensus, and show one column per cluster
  map <- compute_consensus(mixture_model, type = "MAP")
  map$probability <- NULL
  stats::reshape(map, direction = "wide", idvar = "map_ranking",
                 timevar = "cluster", varying = list(as.character(unique(map$cluster))))

  # RUNNING IN PARALLEL
  # Computing Mallows models with different number of mixtures in parallel leads to
  # considerably speedup
  library(parallel)
  cl <- makeCluster(detectCores() - 1)
  n_clusters <- seq(from = 1, to = 10)
  models <- compute_mallows_mixtures(n_clusters = n_clusters,
                                     rankings = sushi_rankings,
                                     include_wcd = TRUE, cl = cl)
  stopCluster(cl)

## End(Not run)
```

compute_posterior_intervals

Compute Posterior Intervals

Description

Compute posterior intervals of parameters of interest.

Usage

```
compute_posterior_intervals(model_fit, ...)
```

Arguments

`model_fit`	An object returned from `compute_mallows`.
`...`	other arguments passed to methods.

Details

This function computes both the Highest Posterior Density Interval (HPDI), which may be discontinuous for bimodal distributions, and the central posterior interval, which is simply defined by the quantiles of the posterior distribution. The HPDI intervals are computed using the `HDInterval` package (Meredith and Kruschke 2018).

References

Meredith M, Kruschke J (2018). *HDInterval: Highest (Posterior) Density Intervals*. R package version 0.2.0, https://CRAN.R-project.org/package=HDInterval.

See Also

`compute_mallows`

Examples

```
# The example datasets potato_visual and potato_weighing contain complete
# rankings of 20 items, by 12 assessors. We first analyse these using the Mallows
# model:
model_fit <- compute_mallows(potato_visual)

# Se the documentation to compute_mallows for how to assess the convergence of the algorithm
# Having chosen burin = 1000, we compute posterior intervals
model_fit$burnin <- 1000
# First we compute the interval for alpha
compute_posterior_intervals(model_fit, parameter = "alpha")
# We can reduce the number decimals
compute_posterior_intervals(model_fit, parameter = "alpha", decimals = 2)
# By default, we get a 95 % interval. We can change that to 99 %.
compute_posterior_intervals(model_fit, parameter = "alpha", level = 0.99)
# We can also compute the posterior interval for the latent ranks rho
compute_posterior_intervals(model_fit, parameter = "rho")

## Not run:
  # Posterior intervals of cluster probabilities
  # We can run a mixture of Mallows models, using the n_clusters argument
  # We use the sushi example data. See the documentation of compute_mallows for a more elaborate
  # example
  model_fit <- compute_mallows(sushi_rankings, n_clusters = 5)
 # Keeping the burnin at 1000, we can compute the posterior intervals of the cluster probabilities
  compute_posterior_intervals(model_fit, burnin = 1000, parameter = "cluster_probs")

## End(Not run)
```

`compute_posterior_intervals.BayesMallows`
Compute posterior intervals

Description

Compute posterior intervals

Usage

```
## S3 method for class 'BayesMallows'
compute_posterior_intervals(
  model_fit,
  burnin = model_fit$burnin,
  parameter = "alpha",
  level = 0.95,
  decimals = 3L,
  ...
)
```

Arguments

`model_fit`	An object returned from `compute_mallows`.
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`parameter`	Character string defining which parameter to compute posterior intervals for. One of `"alpha"`, `"rho"`, or `"cluster_probs"`. Default is `"alpha"`.
`level`	Decimal number in $[0, 1]$ specifying the confidence level. Defaults to `0.95`.
`decimals`	Integer specifying the number of decimals to include in posterior intervals and the mean and median. Defaults to `3`.
`...`	other arguments passed to methods.

See Also

assess_convergence

compute_posterior_intervals.SMCMallows

Compute posterior intervals

Description

Compute posterior intervals

Usage

```
## S3 method for class 'SMCMallows'
compute_posterior_intervals(
  model_fit,
  burnin = model_fit$burnin,
  parameter = "alpha",
  level = 0.95,
  decimals = 3L,
  ...
)
```

Arguments

`model_fit`	An object returned from `compute_mallows`.
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`parameter`	Character string defining which parameter to compute posterior intervals for. One of `"alpha"`, `"rho"`, or `"cluster_probs"`. Default is `"alpha"`.
`level`	Decimal number in $[0, 1]$ specifying the confidence level. Defaults to `0.95`.
`decimals`	Integer specifying the number of decimals to include in posterior intervals and the mean and median. Defaults to `3`.
`...`	other arguments passed to methods.

compute_posterior_intervals_alpha

Compute Posterior Intervals Alpha

Description

posterior confidence intervals

Usage

```
compute_posterior_intervals_alpha(output, nmc, burnin, verbose = FALSE)
```

Arguments

`output`	input
`nmc`	Number of Monte Carlo samples
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`verbose`	if `TRUE`, prints the final output even if the function is assigned to an object. Defaults to `FALSE`.

Author(s)

Anja Stein

`compute_posterior_intervals_rho` *Compute Posterior Intervals Rho*

Description

posterior confidence intervals for rho

Usage

```
compute_posterior_intervals_rho(
  output,
  nmc,
  burnin,
  colnames = NULL,
  verbose = FALSE
)
```

Arguments

`output`	input
`nmc`	Number of Monte Carlo samples
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`colnames`	colnames
`verbose`	if `TRUE`, prints the final output even if the function is assigned to an object. Defaults to `FALSE`.

Author(s)

Anja Stein

compute_rho_consensus *Compute rho consensus*

Description

MAP AND CP consensus ranking estimates

Usage

```
compute_rho_consensus(
  output,
  nmc,
  burnin,
  C,
  type,
  colnames = NULL,
  verbose = FALSE
)
```

Arguments

`output`	input
`nmc`	Number of Monte Carlo samples
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`C`	C
`type`	type
`colnames`	colnames
`verbose`	if `TRUE`, prints the final output even if the function is assigned to an object. Defaults to `FALSE`.

Author(s)

Anja Stein

`correction_kernel` *Correction Kernel*

Description

Function to determine if the augmented ranking is compatible with the new observed partial ranking. If it is not, the we create a new augmentation using the random sampling approach and calculate the augmentation probability.

Usage

```
correction_kernel(observed_ranking, current_ranking, n_items)
```

Arguments

`observed_ranking`
A ranking sequence vector of the observed partial ranking (no missing values) The original incomplete partial ranking is in the rankings data set.

`current_ranking`
A ranking sequence vector of the current augmented ranking (no missing values)

`n_items` Integer is the number of items in a ranking

Value

List containing the proposed 'corrected' augmented ranking that is compatible with the new observed ranking for a user

`correction_kernel_pseudo` *Correction Kernel (pseudolikelihood)*

Description

Function to determine if the augmented ranking is compatible with the new observed partial ranking. If it is not, the we create a new augmentation using the pseudolikelihood approach and calculate the augmentation probability.

Usage

```
correction_kernel_pseudo(
  current_ranking,
  observed_ranking,
  rho,
  alpha,
  n_items,
  metric
)
```

Arguments

`current_ranking`
: An complete rank sequence vector of the proposed augmented ranking obtained from calculate_forward_probability function

`observed_ranking`
: An incomplete rank sequence vector of the original observed incomplete ranking which contains NAs

`rho`
: Numeric vector specifying the consensus ranking

`alpha`
: Numeric value of the scale parameter

`n_items`
: Integer is the number of items in a ranking

`metric`
: A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.

Value

list containing R_obs, the proposed 'corrected' augmented ranking that is compatible with the new observed ranking for a user, and forward_auxiliary_ranking_probability, a numerical value for the probability of correcting the ranking to be compatible with R_obs.

`estimate_partition_function` *Estimate Partition Function*

Description

Estimate the logarithm of the partition function of the Mallows rank model. Choose between the importance sampling algorithm described in (Vitelli et al. 2018) and the IPFP algorithm for computing an asymptotic approximation described in (Mukherjee 2016).

Usage

```
estimate_partition_function(
  method = "importance_sampling",
  alpha_vector,
  n_items,
  metric,
  nmc,
  degree,
  n_iterations,
  K,
  cl = NULL,
  seed = NULL
)
```

Arguments

method	Character string specifying the method to use in order to estimate the logarithm of the partition function. Available options are "importance_sampling" and "asymptotic".
alpha_vector	Numeric vector of α values over which to compute the importance sampling estimate.
n_items	Integer specifying the number of items.
metric	Character string specifying the distance measure to use. Available options are "footrule" and "spearman" when method = "asymptotic" and in addition "cayley", "hamming", "kendall", and "ulam" when method = "importance_sampling".
nmc	Integer specifying the number of Monte Carlo samples to use in the importance sampling. Only used when method = "importance_sampling".
degree	Integer specifying the degree of the polynomial used to estimate $\log(\alpha)$ from the grid of values provided by the importance sampling estimate.
n_iterations	Integer specifying the number of iterations to use in the asymptotic approximation of the partition function. Only used when method = "asymptotic".
K	Integer specifying the parameter K in the asymptotic approximation of the partition function. Only used when method = "asymptotic".
cl	Optional computing cluster used for parallelization, returned from parallel::makeCluster. Defaults to NULL. Only used when method = "importance_sampling".
seed	Optional random number seed.

Value

A vector of length degree which can be supplied to the logz_estimate argument of compute_mallows.

References

Mukherjee S (2016). “Estimation in exponential families on permutations.” *The Annals of Statistics*, **44**(2), 853–875. doi:10.1214/15aos1389.

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). “Probabilistic Preference Learning with the Mallows Rank Model.” *Journal of Machine Learning Research*, **18**(1), 1–49. https://jmlr.org/papers/v18/15-481.html.

Examples

```
## Not run:
  # IMPORTANCE SAMPLING
  # Let us estimate logZ(alpha) for 20 items with Spearman distance
  # We create a grid of alpha values from 0 to 10
  alpha_vector <- seq(from = 0, to = 10, by = 0.5)
  n_items <- 20
  metric <- "spearman"
  degree <- 10

  # We start with 1e3 Monte Carlo samples
```

```
fit1 <- estimate_partition_function(method = "importance_sampling",
                                      alpha_vector = alpha_vector,
                                      n_items = n_items, metric = metric,
                                      nmc = 1e3, degree = degree)
# A vector of polynomial regression coefficients is returned
fit1

# Now let us recompute with 1e4 Monte Carlo samples
fit2 <- estimate_partition_function(method = "importance_sampling",
                                    alpha_vector = alpha_vector,
                                    n_items = n_items, metric = metric,
                                    nmc = 1e4, degree = degree)

# ASYMPTOTIC APPROXIMATION
# We can also compute an estimate using the asymptotic approximation
K <- 20
n_iterations <- 50

fit3 <- estimate_partition_function(method = "asymptotic",
                                    alpha_vector = alpha_vector,
                                    n_items = n_items, metric = metric,
                                    n_iterations = n_iterations,
                                    K = K, degree = degree)

# We write a little function for storing the estimates in a dataframe
powers <- seq(from = 0, to = degree, by = 1)

compute_fit <- function(fit, type){
  do.call(rbind, lapply(alpha_vector, function(alpha){
    data.frame(
      type = type,
      alpha = alpha,
      logz_estimate = sum(alpha^powers * fit)
    )
  }))
}



estimates <- rbind(
  compute_fit(fit1, type = "Importance Sampling 1e3"),
  compute_fit(fit2, type = "Importance Sampling 1e4"),
  compute_fit(fit3, type = "Asymptotic")
  )

# We can now plot the two estimates side-by-side
library(ggplot2)
ggplot(estimates, aes(x = alpha, y = logz_estimate, color = type)) +
  geom_line()
# We see that the two importance sampling estimates, which are unbiased,
# overlap. The asymptotic approximation seems a bit off. It can be worthwhile
# to try different values of n_iterations and K.
```

```
  # When we are happy, we can provide the coefficient vector in the
  # logz_estimate argument to compute_mallows
  # Say we choose to use the importance sampling estimate with 1e4 Monte Carlo samples:
  model_fit <- compute_mallows(potato_visual, metric = "spearman",
                               logz_estimate = fit2)


## End(Not run)
```

expected_dist *Expected value of metrics under a Mallows rank model*

Description

Compute the expectation of several metrics under the Mallows rank model.

Usage

```
expected_dist(alpha, n_items, metric)
```

Arguments

`alpha`	Non-negative scalar specifying the scale (precision) parameter in the Mallows rank model.
`n_items`	Integer specifying the number of items.
`metric`	Character string specifying the distance measure to use. Available options are `"kendall"`, `"cayley"`, `"hamming"`, `"ulam"` for `n_items<=95`, `"footrule"` for `n_items<=50` and `"spearman"` for `n_items<=14`.

Value

A scalar providing the expected value of the `metric` under the Mallows rank model with distance specified by the `metric` argument.

Examples

```
expected_dist(1, 5, metric = "kendall")
expected_dist(2, 6, metric = "cayley")
expected_dist(1.5, 7, metric = "hamming")
expected_dist(5, 30, "ulam")
expected_dist(3.5, 45, "footrule")
expected_dist(4, 10, "spearman")
```

generate_constraints *Generate Constraint Set from Pairwise Comparisons*

Description

This function is relevant when compute_mallows is called repeatedly with the same data, e.g., when determining the number of clusters. It precomputes a list of constraints used internally by the MCMC algorithm, which otherwise would be recomputed each time compute_mallows is called.

Usage

```
generate_constraints(preferences, n_items, cl = NULL)
```

Arguments

preferences	Data frame of preferences. For the case of consistent rankings, preferences should be returned from generate_transitive_closure. For the case of inconsistent preferences, when using an error model as described in Crispino et al. (2019), a dataframe of preferences can be directly provided.
n_items	Integer specifying the number of items.
cl	Optional computing cluster used for parallelization, returned from parallel::makeCluster. Defaults to NULL.

Value

A list which is used internally by the MCMC algorithm.

References

Crispino M, Arjas E, Vitelli V, Barrett N, Frigessi A (2019). "A Bayesian Mallows approach to nontransitive pair comparison data: How human are sounds?" *The Annals of Applied Statistics*, **13**(1), 492–519. doi:10.1214/18aoas1203.

Examples

```
# Here is an example with the beach preference data.
# First, generate the transitive closure.
beach_tc <- generate_transitive_closure(beach_preferences)

# Next, generate an initial ranking.
beach_init_rank <- generate_initial_ranking(beach_tc)

# Then generate the constrain set used intervally by compute_mallows
constr <- generate_constraints(beach_tc, n_items = 15)

# Provide all these elements to compute_mallows
model_fit <- compute_mallows(rankings = beach_init_rank,
preferences = beach_tc, constraints = constr)
```

```
## Not run:
  # The constraints can also be generated in parallel
  library(parallel)
  cl <- makeCluster(detectCores() - 1)
  constr <- generate_constraints(beach_tc, n_items = 15, cl = cl)
  stopCluster(cl)

## End(Not run)
```

generate_initial_ranking

Generate Initial Ranking

Description

Given a consistent set of pairwise preferences, generate a complete ranking of items which is consistent with the preferences.

Usage

```
generate_initial_ranking(
  tc,
  n_items = max(tc[, c("bottom_item", "top_item")]),
  cl = NULL,
  shuffle_unranked = FALSE,
  random = FALSE,
  random_limit = 8L
)
```

Arguments

tc
: A dataframe with pairwise comparisons of `S3` subclass `BayesMallowsTC`, returned from `generate_transitive_closure`.

n_items
: The total number of items. If not provided, it is assumed to equal the largest item index found in `tc`, i.e., `max(tc[,c("bottom_item", "top_item")])`.

cl
: Optional computing cluster used for parallelization, returned from `parallel::makeCluster`. Defaults to `NULL`.

shuffle_unranked
: Logical specifying whether or not to randomly permuted unranked items in the initial ranking. When `shuffle_unranked=TRUE` and `random=FALSE`, all unranked items for each assessor are randomly permuted. Otherwise, the first ordering returned by `igraph::topo_sort()` is returned.

random
: Logical specifying whether or not to use a random initial ranking. Defaults to `FALSE`. Setting this to `TRUE` means that all possible orderings consistent with the stated pairwise preferences are generated for each assessor, and one of them is picked at random.

random_limit Integer specifying the maximum number of items allowed when all possible orderings are computed, i.e., when random=TRUE. Defaults to 8L.

Value

A matrix of rankings which can be given in the rankings argument to compute_mallows.

Note

Setting random=TRUE means that all possible orderings of each assessor's preferences are generated, and one of them is picked at random. This can be useful when experiencing convergence issues, e.g., if the MCMC algorithm does not mixed properly. However, finding all possible orderings is a combinatorial problem, which may be computationally very hard. The result may not even be possible to fit in memory, which may cause the R session to crash. When using this option, please try to increase the size of the problem incrementally, by starting with smaller subsets of the complete data. An example is given below.

As detailed in the documentation to generate_transitive_closure, it is assumed that the items are labeled starting from 1. For example, if a single comparison of the following form is provided, it is assumed that there is a total of 30 items (n_items=30), and the initial ranking is a permutation of these 30 items consistent with the preference 29<30.

assessor	**bottom_item**	**top_item**
1	29	30

If in reality there are only two items, they should be relabeled to 1 and 2, as follows:

assessor	**bottom_item**	**top_item**
1	1	2

Examples

```
# The example dataset beach_preferences contains pairwise preferences of beach.
# We must first generate the transitive closure
beach_tc <- generate_transitive_closure(beach_preferences)

# Next, we generate an initial ranking
beach_init <- generate_initial_ranking(beach_tc)

# Look at the first few rows:
head(beach_init)

# We can add more informative column names in order
# to get nicer posterior plots:
colnames(beach_init) <- paste("Beach", seq(from = 1, to = ncol(beach_init), by = 1))
head(beach_init)

# By default, the algorithm for generating the initial ranking is deterministic.
# It is possible to randomly permuted the unranked items with the argument shuffle_unranked,
# as demonstrated below. This algorithm is computationally efficient, but defaults to FALSE
# for backward compatibility.
set.seed(2233)
beach_init <- generate_initial_ranking(beach_tc, shuffle_unranked = TRUE)
head(beach_init)

# It is also possible to pick a random sample among all topological sorts.
# This requires first enumerating all possible sorts, and might hence be computationally
# demanding. Here is an example, where we reduce the data considerable to speed up computation.
small_tc <- beach_tc[beach_tc$assessor %in% 1:6 &
                       beach_tc$bottom_item %in% 1:4 & beach_tc$top_item %in% 1:4, ]
set.seed(123)
init_small <- generate_initial_ranking(tc = small_tc, n_items = 4, random = TRUE)
# Look at the initial rankings generated
init_small

# For this small dataset, we can also study the effect of setting shuffle_unranked=TRUE
# in more detail. We consider assessors 1 and 2 only.
# First is the deterministic ordering. This one is equal for each run.
generate_initial_ranking(tc = small_tc[small_tc$assessor %in% c(1, 2), ],
                         n_items = 4, shuffle_unranked = FALSE, random = FALSE)
# Next we shuffle the unranked, setting the seed for reproducibility.
# For assessor 1, item 2 is unranked, and by rerunning the code multiple times,
# we see that element (1, 2) indeed changes ranking randomly.
# For assessor 2, item 3 is unranked, and we can also see that this item changes
# ranking randomly when rerunning the function multiple times.
# The ranked items also change their ranking from one random realiziation to another,
# but their relative ordering is constant.
set.seed(123)
generate_initial_ranking(tc = small_tc[small_tc$assessor %in% c(1, 2), ],
                         n_items = 4, shuffle_unranked = TRUE, random = FALSE)


## Not run:
  # We now give beach_init and beach_tc to compute_mallows. We tell compute_mallows
  # to save the augmented data, in order to study the convergence.
  model_fit <- compute_mallows(rankings = beach_init,
                               preferences = beach_tc,
                               nmc = 2000,
                               save_aug = TRUE)

  # We can study the acceptance rate of the augmented rankings
  assess_convergence(model_fit, parameter = "Rtilde")

  # We can also study the posterior distribution of the consensus rank of each beach
  model_fit$burnin <- 500
  plot(model_fit, parameter = "rho", items = 1:15)

## End(Not run)

## Not run:
  # The computations can also be done in parallel
  library(parallel)
  cl <- makeCluster(detectCores() - 1)
  beach_tc <- generate_transitive_closure(beach_preferences, cl = cl)
  beach_init <- generate_initial_ranking(beach_tc, cl = cl)
  stopCluster(cl)

## End(Not run)
```

generate_transitive_closure

Generate Transitive Closure

Description

Generate the transitive closure for a set of consistent pairwise comparisons. The result can be given in the preferences argument to compute_mallows.

Usage

```
generate_transitive_closure(df, cl = NULL)
```

Arguments

df A data frame with one row per pairwise comparison, and columns assessor, top_item, and bottom_item. Each column contains the following:

- assessor is a numeric vector containing the assessor index, or a character vector containing the (unique) name of the assessor.
- bottom_item is a numeric vector containing the index of the item that was disfavored in each pairwise comparison.
- top_item is a numeric vector containing the index of the item that was preferred in each pairwise comparison.

So if we have two assessors and five items, and assessor 1 prefers item 1 to item 2 and item 1 to item 5, while assessor 2 prefers item 3 to item 5, we have the following df:

assessor	**bottom_item**	**top_item**
1	2	1
1	5	1
2	5	3

cl Optional computing cluster used for parallelization, returned from parallel::makeCluster. Defaults to NULL.

Value

A dataframe with the same columns as df, but with its set of rows expanded to include all pairwise preferences implied by the ones stated in df. The returned object has S3 subclass BayesMallowsTC, to indicate that this is the transitive closure.

See Also

generate_initial_ranking

Examples

```
# Let us first consider a simple case with two assessors, where assessor 1
# prefers item 1 to item 2, and item 1 to item 5, while assessor 2 prefers
# item 3 to item 5. We then have the following dataframe of pairwise
# comparisons:
pair_comp <- data.frame(
  assessor = c(1, 1, 2),
  bottom_item = c(2, 5, 5),
  top_item = c(1, 1, 3))
# We then generate the transitive closure of these preferences:
(pair_comp_tc <- generate_transitive_closure(pair_comp))
# In this case, no additional relations we implied by the ones
# stated in pair_comp, so pair_comp_tc has exactly the same rows
# as pair_comp.

# Now assume that assessor 1 also preferred item 5 to item 3, and
# that assessor 2 preferred item 4 to item 3.
pair_comp <- data.frame(
  assessor = c(1, 1, 1, 2, 2),
  bottom_item = c(2, 5, 3, 5, 3),
  top_item = c(1, 1, 5, 3, 4))

# We generate the transitive closure again:
(pair_comp_tc <- generate_transitive_closure(pair_comp))
# We now have one implied relation for each assessor.
# For assessor 1, it is implied that 1 is preferred to 3.
# For assessor 2, it is implied that 4 is preferred to 5.

## Not run:
  # If assessor 1 in addition preferred item 3 to item 1,
  # the preferences would not be consistent. This is not yet supported by compute_mallows,
  # so it causes an error message. It will be supported in a future release of the package.
  # First, we add the inconsistent row to pair_comp
  pair_comp <- rbind(
    pair_comp,
    data.frame(assessor = 1, bottom_item = 1, top_item = 3))

  # This causes an error message and prints out the problematic rankings:
  (pair_comp_tc <- generate_transitive_closure(pair_comp))

## End(Not run)


## Not run:
  # The computations can also be done in parallel
  library(parallel)
  cl <- makeCluster(detectCores() - 1)
  beach_tc <- generate_transitive_closure(beach_preferences, cl = cl)
  stopCluster(cl)

## End(Not run)
```

get_mallows_loglik *Likelihood and log-likelihood evaluation for a Mallows mixture model*

Description

Compute either the likelihood or the log-likelihood value of the Mallows mixture model parameters for a dataset of complete rankings.

Usage

```
get_mallows_loglik(
  rho,
  alpha,
  weights,
  metric,
  rankings,
  obs_freq = NULL,
  log = TRUE
)

lik_db_mix(rho, alpha, weights, metric, rankings, obs_freq = NULL, log = FALSE)
```

Arguments

`rho` A matrix of size `n_clusters` x `n_items` whose rows are permutations of the first n_items integers corresponding to the modal rankings of the Mallows mixture components.

`alpha` A vector of `n_clusters` non-negative scalar specifying the scale (precision) parameters of the Mallows mixture components.

`weights` A vector of `n_clusters` non-negative scalars specifying the mixture weights.

`metric` Character string specifying the distance measure to use. Available options are `"kendall"`, `"cayley"`, `"hamming"`, `"ulam"` for `n_items<=95`, `"footrule"` for `n_items<=50` and `"spearman"` for `n_items<=14`.

`rankings` A matrix with observed rankings in each row.

`obs_freq` A vector of observation frequencies (weights) to apply to each row in `rankings`. This can speed up computation if a large number of assessors share the same rank pattern. Defaults to `NULL`, which means that each row of `rankings` is multiplied by 1. If provided, `obs_freq` must have the same number of elements as there are rows in `rankings`, and `rankings` cannot be `NULL`.

`log` A logical; if TRUE, the log-likelihood value is returned, otherwise its exponential. Default is `TRUE`.

Value

The likelihood or the log-likelihood value corresponding to one or more observed complete rankings under the Mallows mixture rank model with distance specified by the `metric` argument.

Examples

```
# Simulate a sample from a Mallows model with the Kendall distance

n_items <- 5
mydata <- sample_mallows(
  n_samples = 100,
  rho0 = 1:n_items,
  alpha0 = 10,
  metric = "kendall")

# Compute the likelihood and log-likelihood values under the true model...
get_mallows_loglik(
  rho = rbind(1:n_items, 1:n_items),
  alpha = c(10, 10),
  weights = c(0.5, 0.5),
  metric = "kendall",
  rankings = mydata,
  log = FALSE
  )

get_mallows_loglik(
  rho = rbind(1:n_items, 1:n_items),
  alpha = c(10, 10),
  weights = c(0.5, 0.5),
  metric = "kendall",
  rankings = mydata,
  log = TRUE
  )

# or equivalently, by using the frequency distribution
freq_distr <- rank_freq_distr(mydata)
get_mallows_loglik(
  rho = rbind(1:n_items, 1:n_items),
  alpha = c(10, 10),
  weights = c(0.5, 0.5),
  metric = "kendall",
  rankings = freq_distr[, 1:n_items],
  obs_freq = freq_distr[, n_items + 1],
  log = FALSE
  )

get_mallows_loglik(
  rho = rbind(1:n_items, 1:n_items),
  alpha = c(10, 10),
  weights = c(0.5, 0.5),
  metric = "kendall",
  rankings = freq_distr[, 1:n_items],
  obs_freq = freq_distr[, n_items + 1],
  log = TRUE
  )
```

get_sample_probabilities

Get Sample Probabilities

Description

Calculate probability of assigning a set of specific ranks to an specific item given its rank in the consensus ranking

Usage

```
get_sample_probabilities(
  rho_item_rank,
  alpha,
  remaining_set_ranks,
  metric,
  n_items
)
```

Arguments

`rho_item_rank` An integer value rank of an item in the current consensus ranking

`alpha` Numeric value of the scale parameter

`remaining_set_ranks`
A sequence of integer values of the set of possible ranks that we can assign the item

`metric` A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.

`n_items` Integer is the number of items in the consensus ranking

Value

sample_prob_list A numeric sequence of sample probabilities for selecting a specific rank given the current rho_item_rank

label_switching *Checking for Label Switching in the Mallows Mixture Model*

Description

Label switching may sometimes be a problem when running mixture models. The algorithm by Stephens (Stephens 2000), implemented in the `label.switching` package (Papastamoulis 2016), allows assessment of label switching after MCMC. At the moment, this is the only available option in the `BayesMallows` package. The Stephens algorithms requires the individual cluster probabilities of each assessor to be saved in each iteration of the MCMC algorithm. As this potentially requires much memory, the current implementation of `compute_mallows` saves these cluster probabilities to a csv file in each iteration. The example below shows how to perform such a check for label switching in practice.

Beware that this functionality is under development. Later releases might let the user determine the directory and filenames of the csv files.

References

Papastamoulis P (2016). “label.switching: An R Package for Dealing with the Label Switching Problem in MCMC Outputs.” *Journal of Statistical Software, Code Snippets*, **69**(1), 1–24. ISSN 1548-7660, doi:10.18637/jss.v069.c01.

Stephens M (2000). “Dealing with label switching in mixture models.” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **62**(4), 795-809. doi:10.1111/14679868.00265.

Examples

```
## Not run:
  # This example shows how to assess if label switching happens in BayesMallows

  library(BayesMallows)
  # We start by creating a directory in which csv files with individual
  # cluster probabilities should be saved in each step of the MCMC algorithm
  dir.create("./test_label_switch")
  # Next, we go into this directory
  setwd("./test_label_switch/")
  # For comparison, we run compute_mallows with and without saving the cluster
  # probabilities The purpose of this is to assess the time it takes to save
  # the cluster probabilites
  system.time(m <- compute_mallows(rankings = sushi_rankings,
                                   n_clusters = 6, nmc = 2000, save_clus = TRUE,
                                   save_ind_clus = FALSE, verbose = TRUE))
  # With this options, compute_mallows will save cluster_probs2.csv,
  # cluster_probs3.csv, ..., cluster_probs[nmc].csv.
  system.time(m <- compute_mallows(rankings = sushi_rankings, n_clusters = 6,
                                   nmc = 2000, save_clus = TRUE,
                                   save_ind_clus = TRUE, verbose = TRUE))
```

```
 # Next, we check convergence of alpha
 assess_convergence(m)

 # We set the burnin to 1000
 burnin <- 1000

 # Find all files that were saved. Note that the first file saved is cluster_probs2.csv
 cluster_files <- list.files(pattern = "cluster\\_probs[[:digit:]]+\\.csv")

 # Check the size of the files that were saved.
 paste(sum(do.call(file.size, list(cluster_files))) * 1e-6, "MB")

 # Find the iteration each file corresponds to, by extracting its number
 iteration_number <- as.integer(
   gsub("(^[a-zA-Z\\_\\.]*)([0-9]+)([a-zA-Z\\_\\.]+$)", "\\2",
        scluster_files, perl = TRUE))
 # Remove all files before burnin
 file.remove(cluster_files[iteration_number <= burnin])
 # Update the vector of files, after the deletion
 cluster_files <- list.files(pattern = "cluster\\_probs[[:digit:]]+\\.csv")
 # Create 3d array, with dimensions (iterations, assessors, clusters)
 prob_array <- array(dim = c(length(cluster_files), m$n_assessors, m$n_clusters))
 # Read each file, adding to the right element of the array
 for(i in seq_along(cluster_files)){
   prob_array[i, , ] <- as.matrix(
     read.csv(cluster_files[[i]], header = FALSE))
 }


 # Create an integer array of latent allocations, as this is required by label.switching
 z <- subset(m$cluster_assignment, iteration > burnin)
 z$value <- as.integer(gsub("Cluster ", "", z$value))
 z <- stats::reshape(z, direction = "wide", idvar = "iteration", timevar = "assessor")
 z$iteration <- NULL
 z <- as.matrix(z)

 # Now apply Stephen's algorithm
 library(label.switching)
 ls <- label.switching("STEPHENS", z = z, K = m$n_clusters, p = prob_array)

 # Check the proportion of cluster assignments that were switched
 mean(apply(ls$permutations$STEPHENS, 1, function(x) !all.equal(x, seq(1, m$n_clusters))))

 # Remove the rest of the csv files
 file.remove(cluster_files)
 # Move up one directory
 setwd("..")
 # Remove the directory in which the csv files were saved
 file.remove("./test_label_switch/")

## End(Not run)
```

metropolis_hastings_alpha

Metropolis-Hastings Alpha

Description

Function to perform Metropolis-Hastings for new rho under the Mallows model with footrule distance metric!

Usage

```
metropolis_hastings_alpha(
  alpha,
  n_items,
  rankings,
  metric,
  rho,
  logz_estimate,
  alpha_prop_sd,
  lambda,
  alpha_max
)
```

Arguments

`alpha`	Numeric value of the scale parameter
`n_items`	Integer is the number of items in a ranking
`rankings`	the observed rankings, i.e, preference data
`metric`	A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.
`rho`	Numeric vector specifying the current consensus ranking
`logz_estimate`	Estimate grid of log of partition function, computed with `estimate_partition_function` in the BayesMallow R package estimate_partition_function.
`alpha_prop_sd`	Numeric value specifying the standard deviation of the lognormal proposal distribution used for α in the Metropolis-Hastings algorithm. Defaults to `0.1`.
`lambda`	Strictly positive numeric value specifying the rate parameter of the truncated exponential prior distribution of α. Defaults to `0.1`. When `n_cluster > 1`, each mixture component α_c has the same prior distribution.
`alpha_max`	Maximum value of `alpha` in the truncated exponential prior distribution.

Details

`rankings` is a matrix of size $N \times n_items$ of rankings in each row. Alternatively, if N equals 1, `rankings` can be a vector.

Value

alpha or alpha_prime: Numeric value to be used as the proposal of a new alpha

Author(s)

Anja Stein

Examples

```
rho <- c(1, 2, 3, 4, 5, 6)
alpha <- 2
metric <- "footrule"
n_items <- 6
rankings <- sample_mallows(
  rho0 = rho, alpha0 = alpha, n_samples = 10, burnin = 1000, thinning = 500
)
alpha_vector <- seq(from = 0, to = 20, by = 0.1)
iter <- 1e2
degree <- 10

# Estimate the logarithm of the partition function of the Mallows rank model
# using the estimate partition function
logz_estimate <- estimate_partition_function(
  method = "importance_sampling", alpha_vector = alpha_vector,
  n_items = n_items, metric = "footrule", nmc = iter, degree = degree
)

metropolis_hastings_alpha(
  alpha, n_items, rankings, metric, rho, logz_estimate, alpha_prop_sd = 0.5,
  lambda = 0.1, alpha_max = 20
)

metropolis_hastings_alpha(
  alpha, n_items, rankings, metric, rho, logz_estimate,
  alpha_prop_sd = 0.15, lambda = 0.1, alpha_max = 20
)

metropolis_hastings_alpha(
  alpha, n_items, rankings, metric, rho, logz_estimate,
  alpha_prop_sd = 0.5, lambda = 0.15, alpha_max = 20
)

metropolis_hastings_alpha(
  alpha, n_items, rankings, metric, rho, logz_estimate,
  alpha_prop_sd = 0.15, lambda = 0.15, alpha_max = 20
)
```

metropolis_hastings_rho *Metropolis-Hastings Rho*

Description

Function to perform Metropolis-Hastings for new rho under the Mallows model with footrule distance metric!

Usage

```
metropolis_hastings_rho(alpha, n_items, rankings, metric, rho, leap_size)
```

Arguments

alpha	Numeric value of the scale parameter
n_items	Integer is the number of items in a ranking A matrix of size $N \times n_items$ of rankings in each row. Alternatively, if N equals 1, rankings can be a vector.
rankings	A matrix of size $N \times n_items$ of rankings in each row. Alternatively, if N equals 1, rankings can be a vector.
metric	Character string specifying the distance measure to use. Available options are "kendall", "cayley", "hamming", "ulam", "footrule" and "spearman".
rho	A ranking sequence
leap_size	Integer specifying the step size of the leap-and-shift proposal distribution.

Author(s)

Anja Stein

Examples

```
rho <- t(c(1,2,3,4,5,6))
alpha <- 2
metric <- "footrule"
n_items <- 6

metropolis_hastings_rho(
alpha = alpha, n_items = n_items, rankings = rho, metric = metric,
rho = rho, leap_size = 1
)

metropolis_hastings_rho(
alpha = alpha, n_items = n_items, rankings = rho, metric = metric,
rho = rho, leap_size = 2
)

metropolis_hastings_rho(
alpha = alpha, n_items = n_items, rankings = rho, metric = metric,
rho = rho, leap_size = 3
)

rankings <- sample_mallows(
 rho0 = rho, alpha0 = alpha, n_samples = 10, burnin = 1000, thinning = 500
)
metropolis_hastings_rho(
alpha = alpha, n_items = n_items, rankings = rankings, metric = metric,
rho = rho, leap_size = 1
)
```

obs_freq *Observation frequencies in the Bayesian Mallows model*

Description

When more than one assessor have given the exact same rankings or preferences, considerable speed-up can be obtained by providing only the unique set of rankings/preferences to compute_mallows, and instead providing the number of assessors in the obs_freq argument. This topic is illustrated here. See also the function rank_freq_distr for how to easily compute the observation frequencies.

Examples

```
# The first example uses full rankings in the potato_visual dataset, but we assume
# that each row in the data corresponds to between 100 and 500 assessors.
set.seed(1234)
# We start by generating random observation frequencies
obs_freq <- sample(x = seq(from = 100L, to = 500L, by = 1L),
                   size = nrow(potato_visual), replace = TRUE)
# We also create a set of repeated indices, used to extend the matrix rows
repeated_indices <- unlist(Map(function(x, y) rep(x, each = y),
                                seq_len(nrow(potato_visual)), obs_freq))
# The potato_repeated matrix consists of all rows repeated corresponding to
# the number of assessors in the obs_freq vector. This is how a large dataset
# would look like without using the obs_freq argument
potato_repeated <- potato_visual[repeated_indices, ]

# We now first compute the Mallows model using obs_freq
# This takes about 0.2 seconds
system.time({
 m_obs_freq <- compute_mallows(rankings = potato_visual, obs_freq = obs_freq, nmc = 10000)
})
# Next we use the full ranking matrix
# This takes about 11.3 seconds, about 50 times longer!
## Not run:
system.time({
  m_rep <- compute_mallows(rankings = potato_repeated, nmc = 10000)
```

```
})

  # We set the burnin to 2000 for both
  m_obs_freq$burnin <- 2000
  m_rep$burnin <- 2000

  # Note that the MCMC algorithms did not run with the same
  # random number seeds in these two experiments, but still
  # the posterior distributions look similar
  plot(m_obs_freq, burnin = 2000, "alpha")
  plot(m_rep, burnin = 2000, "alpha")

  plot(m_obs_freq, burnin = 2000, "rho", items = 1:4)
  plot(m_rep, burnin = 2000, "rho", items = 1:4)

## End(Not run)


# Next we repeated the exercise with the pairwise preference data
# in the beach dataset. Note that we first must compute the
# transitive closure for each participant. If two participants
# have provided different preferences with identical transitive closure,
# then we can treat them as identical
beach_tc <- generate_transitive_closure(beach_preferences)
# Next, we confirm that each participant has a unique transitive closure
# We do this by sorting first by top_item and then by bottom_item,
# and then concatenating, whereupon we check how many participants there
# are for each unique concatenation
# This returns zero rows, so there are no participants with the same transitive closure
beach_tc <- beach_tc[order(beach_tc$assessor, beach_tc$top_item,
                           beach_tc$bottom_item), ]
aggr_df <- do.call(rbind, lapply(split(beach_tc, f = beach_tc$assessor), function(x){
  x$concat_ranks <- paste(c(x$bottom_item, x$top_item), collapse = ",")
  x
}))

aggr_df <- aggregate(list(num_assessors = aggr_df$assessor),
                     aggr_df[, "concat_ranks", drop = FALSE],
                     FUN = function(x) length(unique(x)))

nrow(aggr_df[aggr_df$num_assessors > 1, , drop = FALSE])

# We now illustrate the weighting procedure by assuming that there are
# more than one assessor per unique transitive closure. We generate an
# obs_freq vector such that each unique transitive closure is repeated 1-4 times.
set.seed(9988)
obs_freq <- sample(x = 1:4, size = length(unique(beach_preferences$assessor)), replace = TRUE)

# Next, we create a new hypthetical beach_preferences dataframe where each
# assessor is replicated 1-4 times
beach_pref_rep <- do.call(
  rbind,
  lapply(split(beach_preferences, f = beach_preferences$assessor),
         function(dd){
           ret <- merge(
             dd,
             data.frame(
               new_assessor = seq_len(obs_freq[unique(dd$assessor)])
               ), all = TRUE)
           ret$assessor <- paste(ret$assessor, ret$new_assessor, sep = ”,”)
           ret$new_assessor <- NULL
           ret
           }))

# We generate transitive closure for these preferences
beach_tc_rep <- generate_transitive_closure(beach_pref_rep)
# We can check that the number of unique assessors is now larger,
# and equal to the sum of obs_freq
sum(obs_freq)
length(unique(beach_tc_rep$assessor))

# We generate the initial rankings for the repeated and the ”unrepeated”
# data
beach_rankings <- generate_initial_ranking(beach_tc, n_items = 15)
beach_rankings_rep <- generate_initial_ranking(beach_tc_rep, n_items = 15)

## Not run:
# We then run the Bayesian Mallows rank model, first for the
# unrepeated data with a obs_freq argument. This takes about 1.9 seconds
system.time({
  model_fit_obs_freq <- compute_mallows(rankings = beach_rankings,
                                       preferences = beach_tc,
                                       obs_freq = obs_freq,
                                       save_aug = TRUE,
                                       nmc = 10000)

})

# Next for the repeated data. This takes about 4.8 seconds.
system.time({
  model_fit_rep <- compute_mallows(rankings = beach_rankings_rep,
                                   preferences = beach_tc_rep,
                                   save_aug = TRUE,
                                   nmc = 10000)

})

# As demonstrated here, using a obs_freq argument to exploit patterns in data
# where multiple assessors have given identical rankings or preferences, may
# lead to considerable speedup.


## End(Not run)
```

plot.BayesMallows *Plot Posterior Distributions*

Description

Plot posterior distributions of the parameters of the Mallows Rank model.

Usage

```
## S3 method for class 'BayesMallows'
plot(x, burnin = x$burnin, parameter = "alpha", items = NULL, ...)
```

Arguments

x	An object of type BayesMallows, returned from compute_mallows.
burnin	A numeric value specifying the number of iterations to discard as burn-in. Defaults to x$burnin, and must be provided if x$burnin does not exist. See assess_convergence.
parameter	Character string defining the parameter to plot. Available options are "alpha", "rho", "cluster_probs", "cluster_assignment", and "theta".
items	The items to study in the diagnostic plot for rho. Either a vector of item names, corresponding to x$items or a vector of indices. If NULL, five items are selected randomly. Only used when parameter = "rho".
...	Other arguments passed to plot (not used).

Examples

```
# The example datasets potato_visual and potato_weighing contain complete
# rankings of 20 items, by 12 assessors. We first analyse these using the Mallows
# model:
model_fit <- compute_mallows(potato_visual)

# Se the documentation to compute_mallows for how to assess the convergence
# of the algorithm
# We set the burnin = 1000
model_fit$burnin <- 1000
# By default, the scale parameter "alpha" is plotted
plot(model_fit)
## Not run:
  # We can also plot the latent rankings "rho"
  plot(model_fit, parameter = "rho")
  # By default, a random subset of 5 items are plotted
  # Specify which items to plot in the items argument.
  plot(model_fit, parameter = "rho",
       items = c(2, 4, 6, 9, 10, 20))
  # When the ranking matrix has column names, we can also
  # specify these in the items argument.
  # In this case, we have the following names:
```

```
  colnames(potato_visual)
  # We can therefore get the same plot with the following call:
  plot(model_fit, parameter = "rho",
       items = c("P2", "P4", "P6", "P9", "P10", "P20"))

## End(Not run)

## Not run:
  # Plots of mixture parameters:
  # We can run a mixture of Mallows models, using the n_clusters argument
 # We use the sushi example data. See the documentation of compute_mallows for a more elaborate
  # example
  model_fit <- compute_mallows(sushi_rankings, n_clusters = 5, save_clus = TRUE)
  model_fit$burnin <- 1000
  # We can then plot the posterior distributions of the cluster probabilities
  plot(model_fit, parameter = "cluster_probs")
  # We can also get a cluster assignment plot, showing the assessors along the horizontal
  # axis and the clusters along the vertical axis. The color show the probability
  # of belonging to each clusters. The assessors are sorted along the horizontal
  # axis according to their maximum a posterior cluster assignment. This plot
  # illustrates the posterior uncertainty in cluster assignments.
  plot(model_fit, parameter = "cluster_assignment")
  # See also ?assign_cluster for a function which returns the cluster assignment
  # back in a dataframe.

## End(Not run)
```

plot_alpha_posterior *Plot Alpha Posterior*

Description

posterior for alpha

Usage

```
plot_alpha_posterior(output, nmc, burnin)
```

Arguments

output	input
nmc	Number of Monte Carlo samples
burnin	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See assess_convergence.

Author(s)

Anja Stein

---

plot_elbow *Plot Within-Cluster Sum of Distances*

Description

Plot the within-cluster sum of distances from the corresponding cluster consensus for different number of clusters. This function is useful for selecting the number of mixture.

Usage

```
plot_elbow(..., burnin = NULL)
```

Arguments

...	One or more objects returned from compute_mallows, separated by comma, or a list of such objects. Typically, each object has been run with a different number of mixtures, as specified in the n_clusters argument to compute_mallows.
burnin	The number of iterations to discard as burnin. Either a vector of numbers, one for each model, or a single number which is taken to be the burnin for all models. If each model provided has a burnin element, then this is taken as the default.

Value

A boxplot with the number of clusters on the horizontal axis and the with-cluster sum of distances on the vertical axis.

See Also

compute_mallows

Examples

```
# DETERMINING THE NUMBER OF CLUSTERS IN THE SUSHI EXAMPLE DATA
## Not run:
  # Let us look at any number of clusters from 1 to 10
  # We use the convenience function compute_mallows_mixtures
  n_clusters <- seq(from = 1, to = 10)
  models <- compute_mallows_mixtures(n_clusters = n_clusters,
                                     rankings = sushi_rankings,
                                     include_wcd = TRUE)
  # models is a list in which each element is an object of class BayesMallows,
  # returned from compute_mallows
  # We can create an elbow plot
  plot_elbow(models, burnin = 1000)
  # We then select the number of cluster at a point where this plot has
  # an "elbow", e.g., n_clusters = 5.

  # Having chosen the number of clusters, we can now study the final model
  # Rerun with 5 clusters, now setting save_clus = TRUE to get cluster assignments
  mixture_model <- compute_mallows(rankings = sushi_rankings, n_clusters = 5,
                                   include_wcd = TRUE, save_clus = TRUE)
  # Delete the models object to free some memory
  rm(models)
  # Set the burnin
  mixture_model$burnin <- 1000
  # Plot the posterior distributions of alpha per cluster
  plot(mixture_model)
  # Compute the posterior interval of alpha per cluster
  compute_posterior_intervals(mixture_model, parameter = "alpha")
  # Plot the posterior distributions of cluster probabilities
  plot(mixture_model, parameter = "cluster_probs")
  # Plot the posterior probability of cluster assignment
  plot(mixture_model, parameter = "cluster_assignment")
  # Plot the posterior distribution of "tuna roll" in each cluster
  plot(mixture_model, parameter = "rho", items = "tuna roll")
  # Compute the cluster-wise CP consensus, and show one column per cluster
  cp <- compute_consensus(mixture_model, type = "CP")
  cp$cumprob <- NULL
  stats::reshape(cp, direction = "wide", idvar = "ranking",
                 timevar = "cluster", varying = list(as.character(unique(cp$cluster))))

  # Compute the MAP consensus, and show one column per cluster
  map <- compute_consensus(mixture_model, type = "MAP")
  map$probability <- NULL
  stats::reshape(map, direction = "wide", idvar = "map_ranking",
                 timevar = "cluster", varying = list(as.character(unique(map$cluster))))

  # RUNNING IN PARALLEL
  # Computing Mallows models with different number of mixtures in parallel leads to
  # considerably speedup
  library(parallel)
  cl <- makeCluster(detectCores() - 1)
  n_clusters <- seq(from = 1, to = 10)
  models <- compute_mallows_mixtures(n_clusters = n_clusters,
                                     rankings = sushi_rankings,
                                     include_wcd = TRUE, cl = cl)
  stopCluster(cl)

## End(Not run)
```

plot_rho_posterior *Plot the posterior for rho for each item*

Description

Plot the posterior for rho for each item

Usage

```
plot_rho_posterior(output, nmc, burnin, C, colnames = NULL, items = NULL)
```

Arguments

`output`	input
`nmc`	Number of Monte Carlo samples
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`
`C`	Number of cluster
`colnames`	A vector of item names. If NULL, we generate generic names for the items in the ranking.
`items`	Either a vector of item names, or a vector of indices. If NULL, five items are selected randomly.

`plot_top_k` *Plot Top-k Rankings with Pairwise Preferences*

Description

Plot the posterior probability, per item, of being ranked among the top-k for each assessor. This plot is useful when the data take the form of pairwise preferences.

Usage

```
plot_top_k(
  model_fit,
  burnin = model_fit$burnin,
  k = 3,
  rel_widths = c(model_fit$n_clusters, 10)
)
```

Arguments

`model_fit`	An object of type `BayesMallows`, returned from `compute_mallows`.
`burnin`	A numeric value specifying the number of iterations to discard as burn-in. Defaults to `model_fit$burnin`, and must be provided if `model_fit$burnin` does not exist. See `assess_convergence`.
`k`	Integer specifying the k in top-k.
`rel_widths`	The relative widths of the plots of `rho` per cluster and the plot of assessors, respectively. This argument is passed on to `plot_grid`.

See Also

predict_top_k

Examples

```
## Not run:
  # We use the example dataset with beach preferences. Se the documentation to
  # compute_mallows for how to assess the convergence of the algorithm
  # We need to save the augmented data, so setting this option to TRUE
  model_fit <- compute_mallows(preferences = beach_preferences,
                               save_aug = TRUE)
  # We set burnin = 1000
  model_fit$burnin <- 1000
  # By default, the probability of being top-3 is plotted
  plot_top_k(model_fit)
  # We can also plot the probability of being top-5, for each item
  plot_top_k(model_fit, k = 5)
  # We get the underlying numbers with predict_top_k
  probs <- predict_top_k(model_fit)
  # To find all items ranked top-3 by assessors 1-3 with probability more than 80 %,
  # we do
  subset(probs, assessor %in% 1:3 & prob > 0.8)


## End(Not run)
```

---

potato_true_ranking *True ranking of the weights of 20 potatoes.*

Description

True ranking of the weights of 20 potatoes.

Usage

```
potato_true_ranking
```

Format

An object of class `numeric` of length 20.

References

Liu Q, Crispino M, Scheel I, Vitelli V, Frigessi A (2019). “Model-Based Learning from Preference Data.” *Annual Review of Statistics and Its Application*, **6**(1). doi:10.1146/annurevstatistics031017-100213.

potato_visual — *Result of ranking potatoes by weight, where the assessors were only allowed to inspected the potatoes visually. 12 assessors ranked 20 potatoes.*

Description

Result of ranking potatoes by weight, where the assessors were only allowed to inspected the potatoes visually. 12 assessors ranked 20 potatoes.

Usage

```
potato_visual
```

Format

An object of class `matrix` (inherits from `array`) with 12 rows and 20 columns.

References

Liu Q, Crispino M, Scheel I, Vitelli V, Frigessi A (2019). "Model-Based Learning from Preference Data." *Annual Review of Statistics and Its Application*, **6**(1). doi:10.1146/annurevstatistics031017-100213.

potato_weighing — *Result of ranking potatoes by weight, where the assessors were allowed to lift the potatoes. 12 assessors ranked 20 potatoes.*

Description

Result of ranking potatoes by weight, where the assessors were allowed to lift the potatoes. 12 assessors ranked 20 potatoes.

Usage

```
potato_weighing
```

Format

An object of class `matrix` (inherits from `array`) with 12 rows and 20 columns.

References

Liu Q, Crispino M, Scheel I, Vitelli V, Frigessi A (2019). "Model-Based Learning from Preference Data." *Annual Review of Statistics and Its Application*, **6**(1). doi:10.1146/annurevstatistics031017-100213.

predict_top_k *Predict Top-k Rankings with Pairwise Preferences*

Description

Predict the posterior probability, per item, of being ranked among the top-k for each assessor. This is useful when the data take the form of pairwise preferences.

Usage

```
predict_top_k(model_fit, burnin = model_fit$burnin, k = 3)
```

Arguments

model_fit	An object of type BayesMallows, returned from compute_mallows.
burnin	A numeric value specifying the number of iterations to discard as burn-in. Defaults to model_fit$burnin, and must be provided if model_fit$burnin does not exist. See assess_convergence.
k	Integer specifying the k in top-k.

Value

A dataframe with columns assessor, item, and prob, where each row states the probability that the given assessor rates the given item among top-k.

See Also

plot_top_k

Examples

```
## Not run:
  # We use the example dataset with beach preferences. Se the documentation to
  # compute_mallows for how to assess the convergence of the algorithm
  # We need to save the augmented data, so setting this option to TRUE
  model_fit <- compute_mallows(preferences = beach_preferences,
                               save_aug = TRUE)
  # We set burnin = 1000
  model_fit$burnin <- 1000
  # By default, the probability of being top-3 is plotted
  plot_top_k(model_fit)
  # We can also plot the probability of being top-5, for each item
  plot_top_k(model_fit, k = 5)
  # We get the underlying numbers with predict_top_k
  probs <- predict_top_k(model_fit)
  # To find all items ranked top-3 by assessors 1-3 with probability more than 80 %,
  # we do
  subset(probs, assessor %in% 1:3 & prob > 0.8)
```

```
## End(Not run)
```

print.BayesMallows *Print Method for BayesMallows Objects*

Description

The default print method for a BayesMallows object.

Usage

```
## S3 method for class 'BayesMallows'
print(x, ...)
```

Arguments

x	An object of type BayesMallows, returned from compute_mallows.
...	Other arguments passed to print (not used).

print.BayesMallowsMixtures *Print Method for BayesMallowsMixtures Objects*

Description

The default print method for a BayesMallowsMixtures object.

Usage

```
## S3 method for class 'BayesMallowsMixtures'
print(x, ...)
```

Arguments

x	An object of type BayesMallowsMixtures, returned from compute_mallows_mixtures.
...	Other arguments passed to print (not used).

rank_conversion *Convert between ranking and ordering.*

Description

create_ranking takes a vector or matrix of ordered items orderings and returns a corresponding vector or matrix of ranked items. create_ordering takes a vector or matrix of rankings rankings and returns a corresponding vector or matrix of ordered items.

Usage

```
create_ranking(orderings)

create_ordering(rankings)
```

Arguments

orderings	A vector or matrix of ordered items. If a matrix, it should be of size N times n, where N is the number of samples and n is the number of items.
rankings	A vector or matrix of ranked items. If a matrix, it should be N times n, where N is the number of samples and n is the number of items.

Value

A vector or matrix of rankings. Missing orderings coded as NA are propagated into corresponding missing ranks and vice versa.

Functions

- create_ranking(): Convert from ordering to ranking.
- create_ordering(): Convert from ranking to ordering.

Examples

```
# A vector of ordered items.
orderings <- c(5, 1, 2, 4, 3)
# Get ranks
rankings <- create_ranking(orderings)
# rankings is c(2, 3, 5, 4, 1)
# Finally we convert it backed to an ordering.
orderings_2 <- create_ordering(rankings)
# Confirm that we get back what we had
all.equal(orderings, orderings_2)

# Next, we have a matrix with N = 19 samples
# and n = 4 items
set.seed(21)
N <- 10
```

```
n <- 4
orderings <- t(replicate(N, sample.int(n)))
# Convert the ordering to ranking
rankings <- create_ranking(orderings)
# Now we try to convert it back to an ordering.
orderings_2 <- create_ordering(rankings)
# Confirm that we get back what we had
all.equal(orderings, orderings_2)
```

`rank_distance` *Distance between a set of rankings and a given rank sequence*

Description

Compute the distance between a matrix of rankings and a rank sequence.

Usage

```
rank_distance(rankings, rho, metric, obs_freq = 1)
```

Arguments

`rankings` A matrix of size $N \times n_i tems$ of rankings in each row. Alternatively, if N equals 1, `rankings` can be a vector.

`rho` A ranking sequence.

`metric` Character string specifying the distance measure to use. Available options are `"kendall"`, `"cayley"`, `"hamming"`, `"ulam"`, `"footrule"` and `"spearman"`.

`obs_freq` Vector of observation frequencies of length N, or of length 1, which means that all ranks are given the same weight. Defaults to 1.

Details

The implementation of Cayley distance is based on a `C++` translation of `Rankcluster::distCayley` (Grimonprez and Jacques 2016).

Value

A vector of distances according to the given `metric`.

References

Grimonprez Q, Jacques J (2016). *Rankcluster: Model-Based Clustering for Multivariate Partial Ranking Data*. R package version 0.94, https://CRAN.R-project.org/package=Rankcluster.

Examples

```
# Distance between two vectors of rankings:
rank_distance(1:5, 5:1, metric = "kendall")
rank_distance(c(2, 4, 3, 6, 1, 7, 5), c(3, 5, 4, 7, 6, 2, 1), metric = "cayley")
rank_distance(c(4, 2, 3, 1), c(3, 4, 1, 2), metric = "hamming")
rank_distance(c(1, 3, 5, 7, 9, 8, 6, 4, 2), c(1, 2, 3, 4, 9, 8, 7, 6, 5), "ulam")
rank_distance(c(8, 7, 1, 2, 6, 5, 3, 4), c(1, 2, 8, 7, 3, 4, 6, 5), "footrule")
rank_distance(c(1, 6, 2, 5, 3, 4), c(4, 3, 5, 2, 6, 1), "spearman")

# Difference between a metric and a vector
# We set the burn-in and thinning too low for the example to run fast
data0 <- sample_mallows(rho0 = 1:10, alpha = 20, n_samples = 1000,
                        burnin = 10, thinning = 1)

rank_distance(rankings = data0, rho = 1:10, metric = "kendall")
```

rank_freq_distr *Frequency distribution of the ranking sequences*

Description

Construct the frequency distribution of the distinct ranking sequences from the dataset of the individual rankings. This can be of interest in itself, but also used to speed up computation by providing the obs_freq argument to compute_mallows.

Usage

```
rank_freq_distr(rankings)
```

Arguments

rankings	A matrix with the individual rankings in each row.

Value

Numeric matrix with the distinct rankings in each row and the corresponding frequencies indicated in the last (n_items+1)-th column.

Examples

```
# Create example data. We set the burn-in and thinning very low
# for the sampling to go fast
data0 <- sample_mallows(rho0 = 1:5, alpha = 10, n_samples = 1000,
                        burnin = 10, thinning = 1)
# Find the frequency distribution
rank_freq_distr(rankings = data0)
```

sample_dataset *A synthetic 3D matrix generated using the sample_mallows function*

Description

A synthetic 3D matrix (`n_users`, `n_items`, `Time`) generated using the sample_mallows function. These are test datasets used to run the SMC-Mallows framework for the cases where we know all of the users in our system and their original ranking information are partial rankings. However at some point in time, we observe extra information about an existing user in the form of a rank for an item that was previously not known (`NA`). These datasets are very contrived as the first time step (`sample_dataset[, , 1]`) we observed the top `m / 2` items from each user, where `m` is the number of items in a ranking. Then, as we increase the time, we observe the next top ranked item from one user at a time, then the next top ranked item, and so on until we have a complete dataset at `sample_dataset[, , Time]`.

Usage

```
sample_dataset
```

Format

An object of class `array` of dimension 10 x 6 x 31.

References

https://github.com/anjastein/SMC-Mallows/tree/main/data

sample_mallows *Random Samples from the Mallows Rank Model*

Description

Generate random samples from the Mallows Rank Model (Mallows 1957) with consensus ranking ρ and scale parameter α. The samples are obtained by running the Metropolis-Hastings algorithm described in Appendix C of Vitelli et al. (2018).

Usage

```
sample_mallows(
  rho0,
  alpha0,
  n_samples,
  leap_size = max(1L, floor(n_items/5)),
  metric = "footrule",
  diagnostic = FALSE,
  burnin = ifelse(diagnostic, 0, 1000),
  thinning = ifelse(diagnostic, 1, 1000),
  items_to_plot = NULL,
  max_lag = 1000L
)
```

Arguments

`rho0` Vector specifying the latent consensus ranking in the Mallows rank model.

`alpha0` Scalar specifying the scale parameter in the Mallows rank model.

`n_samples` Integer specifying the number of random samples to generate. When `diagnostic = TRUE`, this number must be larger than 1.

`leap_size` Integer specifying the step size of the leap-and-shift proposal distribution.

`metric` Character string specifying the distance measure to use. Available options are `"footrule"` (default), `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`. See also the `rmm` function in the `PerMallows` package (Irurozki et al. 2016) for sampling from the Mallows model with Cayley, Hamming, Kendall, and Ulam distances.

`diagnostic` Logical specifying whether to output convergence diagnostics. If `TRUE`, a diagnostic plot is printed, together with the returned samples.

`burnin` Integer specifying the number of iterations to discard as burn-in. Defaults to 1000 when `diagnostic = FALSE`, else to 0.

`thinning` Integer specifying the number of MCMC iterations to perform between each time a random rank vector is sampled. Defaults to 1000 when `diagnostic = FALSE`, else to 1.

`items_to_plot` Integer vector used if `diagnostic = TRUE`, in order to specify the items to plot in the diagnostic output. If not provided, 5 items are picked at random.

`max_lag` Integer specifying the maximum lag to use in the computation of autocorrelation. Defaults to 1000L. This argument is passed to `stats::acf`. Only used when `diagnostic = TRUE`.

References

Irurozki E, Calvo B, Lozano JA (2016). “PerMallows: An R Package for Mallows and Generalized Mallows Models.” *Journal of Statistical Software*, **71**(12), 1–30. doi:10.18637/jss.v071.i12.

Mallows CL (1957). “Non-Null Ranking Models. I.” *Biometrika*, **44**(1/2), 114–130.

Vitelli V, Sørensen, Crispino M, Arjas E, Frigessi A (2018). “Probabilistic Preference Learning with the Mallows Rank Model.” *Journal of Machine Learning Research*, **18**(1), 1–49. `https://jmlr.org/papers/v18/15-481.html`.

Examples

```
# Sample 100 random rankings from a Mallows distribution with footrule distance
set.seed(1)
# Number of items
n_items <- 15
```

```
# Set the consensus ranking
rho0 <- seq(from = 1, to = n_items, by = 1)
# Set the scale
alpha0 <- 10
# Number of samples
n_samples <- 100
# We first do a diagnostic run, to find the thinning and burnin to use
# We set n_samples to 1000, in order to run 1000 diagnostic iterations.
test <- sample_mallows(rho0 = rho0, alpha0 = alpha0, diagnostic = TRUE,
                       n_samples = 1000, burnin = 1, thinning = 1)
# When items_to_plot is not set, 5 items are picked at random. We can change this.
# We can also reduce the number of lags computed in the autocorrelation plots
test <- sample_mallows(rho0 = rho0, alpha0 = alpha0, diagnostic = TRUE,
                       n_samples = 1000, burnin = 1, thinning = 1,
                       items_to_plot = c(1:3, 10, 15), max_lag = 500)
# From the autocorrelation plot, it looks like we should use
# a thinning of at least 200. We set thinning = 1000 to be safe,
# since the algorithm in any case is fast. The Markov Chain
# seems to mix quickly, but we set the burnin to 1000 to be safe.
# We now run sample_mallows again, to get the 100 samples we want:
samples <- sample_mallows(rho0 = rho0, alpha0 = alpha0, n_samples = 100,
                          burnin = 1000, thinning = 1000)
# The samples matrix now contains 100 rows with rankings of 15 items.
# A good diagnostic, in order to confirm that burnin and thinning are set high
# enough, is to run compute_mallows on the samples
model_fit <- compute_mallows(samples, nmc = 10000)
# The highest posterior density interval covers alpha0 = 10.
compute_posterior_intervals(model_fit, burnin = 2000, parameter = "alpha")
```

smc_mallows_new_users *SMC-Mallows New Users*

Description

Function to perform resample-move SMC algorithm where we receive new users with complete rankings at each time step

Usage

```
smc_mallows_new_users(
  R_obs,
  type,
  n_items,
  metric,
  leap_size,
  N,
  Time,
  mcmc_kernel_app,
  num_new_obs,
  alpha_prop_sd = 1,
  lambda = 1,
  alpha_max = 1,
  alpha = 0,
  aug_method = "random",
  logz_estimate = NULL,
  verbose = FALSE
)

smc_mallows_new_users_partial(
  R_obs,
  n_items,
  metric,
  leap_size,
  N,
  Time,
  logz_estimate,
  mcmc_kernel_app,
  num_new_obs,
  alpha_prop_sd,
  lambda,
  alpha_max,
  aug_method,
  verbose = FALSE
)

smc_mallows_new_users_complete(
  R_obs,
  n_items,
  metric,
  leap_size,
  N,
  Time,
  mcmc_kernel_app,
  num_new_obs,
  alpha_prop_sd,
  lambda,
  alpha_max,
  logz_estimate,
  verbose = FALSE
)

smc_mallows_new_users_partial_alpha_fixed(
  R_obs,
  n_items,
  metric,
  leap_size,
  N,
  Time,
  logz_estimate,
  mcmc_kernel_app,
  num_new_obs,
  aug_method,
  alpha
)
```

Arguments

`R_obs` Matrix containing the full set of observed rankings of size n_assessors by n_items

`type` One of `"complete"`, `"partial"`, or `"partial_alpha_fixed"`.

`n_items` Integer is the number of items in a ranking

`metric` A character string specifying the distance metric to use in the Bayesian Mallows Model. Available options are `"footrule"`, `"spearman"`, `"cayley"`, `"hamming"`, `"kendall"`, and `"ulam"`.

`leap_size` leap_size Integer specifying the step size of the leap-and-shift proposal distribution

`N` Integer specifying the number of particles

`Time` Integer specifying the number of time steps in the SMC algorithm

`mcmc_kernel_app` Integer value for the number of applications we apply the MCMC move kernel

`num_new_obs` Integer value for the number of new observations (complete rankings) for each time step

`alpha_prop_sd` Numeric value specifying the standard deviation of the lognormal proposal distribution used for α in the Metropolis-Hastings algorithm. Defaults to `0.1`.

`lambda` Strictly positive numeric value specifying the rate parameter of the truncated exponential prior distribution of α. Defaults to `0.1`. When `n_cluster > 1`, each mixture component α_c has the same prior distribution.

`alpha_max` Maximum value of `alpha` in the truncated exponential prior distribution.

`alpha` A numeric value of the scale parameter which is known and fixed.

`aug_method` A character string specifying the approach for filling in the missing data, options are "pseudolikelihood" or "random".

`logz_estimate` Estimate of the partition function, computed with `estimate_partition_function` in the BayesMallow R package estimate_partition_function.

`verbose` Logical specifying whether to print out the progress of the SMC-Mallows algorithm. Defaults to `FALSE`.

Value

a set of particles each containing a value of rho and alpha

Functions

- `smc_mallows_new_users_partial()`: Deprecated function for `smc_mallows_new_users_partial`.
- `smc_mallows_new_users_complete()`: Deprecated function for `smc_mallows_new_users_complete`.
- `smc_mallows_new_users_partial_alpha_fixed()`: Deprecated function for `smc_mallows_new_users_partial_al`

Examples

```
# Generate basic elements
data <- sushi_rankings[1:100, ]
n_items <- ncol(sushi_rankings)
metric <- "footrule"
num_new_obs <- 10
logz_estimate <- estimate_partition_function(
  method = "importance_sampling",
  alpha_vector = seq(from = 0, to = 15, by = 0.1),
  n_items = n_items, metric = metric, nmc = 1e2, degree = 10
)

# Calculating rho and alpha samples
samples <- smc_mallows_new_users(
  R_obs = data, type = "complete", n_items = n_items, metric = metric,
  leap_size = floor(n_items / 5), N = 100, Time = nrow(data) / num_new_obs,
  mcmc_kernel_app = 5, logz_estimate = logz_estimate,
  alpha_prop_sd = 0.1, lambda = 0.001, alpha_max = 1e6,
  num_new_obs = num_new_obs, verbose = TRUE
)

# Studying the structure of the output
str(samples)
```

`smc_processing` *SMC Processing*

Description

SMC Processing

Usage

```
smc_processing(output, colnames = NULL)
```

Arguments

`output`	input
`colnames`	colnames

Author(s)

Anja Stein

`sushi_rankings` *Sushi Rankings*

Description

Complete rankings of 10 types of sushi from 5000 assessors (Kamishima 2003).

Usage

```
sushi_rankings
```

Format

An object of class `matrix` (inherits from `array`) with 5000 rows and 10 columns.

References

Kamishima T (2003). "Nantonac Collaborative Filtering: Recommendation Based on Order Responses." In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 583–588.

Index