Package: BayesMallows (via r-universe)

October 29, 2026

Type Package

Title Bayesian Preference Learning with the Mallows Rank Model

Version 2.2.7

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>; Sorensen et al., R Journal, 2020 <doi:10.32614/RJ-2020-026>; Stein, PhD Thesis, 2023 <https://eprints.lancs.ac.uk/id/eprint/195759>). 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, https://ocbe-uio.github.io/BayesMallows/

BugReports https://github.com/ocbe-uio/BayesMallows/issues

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

Depends R (>= 3.5.0)

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

LinkingTo Rcpp, RcppArmadillo, testthat

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

VignetteBuilder knitr, rmarkdown

RdMacros Rdpack

Config/testthat/edition 3

NeedsCompilation yes

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

Repository https://cran.r-universe.dev

Date/Publication 2026-01-20 11:20:08 UTC

RemoteUrl https://github.com/cran/BayesMallows

RemoteRef HEAD

RemoteSha 4b51f95bd57ff9f75e2bebe156da6460c342587e

Contents

assess_convergence . . . 3
assign_cluster . . . 4
beach_preferences . . . 5
bernoulli_data . . . 6
burnin . . . 6
burnin<- . . . 8
cluster_data . . . 9
compute_consensus . . . 9
compute_exact_partition_function . . . 12
compute_expected_distance . . . 13
compute_mallows . . . 14
compute_mallows_mixtures . . . 19
compute_mallows_sequentially . . . 23
compute_observation_frequency . . . 25
compute_posterior_intervals . . . 26
compute_rank_distance . . . 28
create_ranking . . . 29
estimate_partition_function . . . 30
get_acceptance_ratios . . . 32
get_cardinalities . . . 34
get_mallows_loglik . . . 35

get_transitive_closure . . . 37
heat_plot . . . 38
plot.BayesMallows . . . 39
plot.SMCMallows . . . 40
plot_elbow . . . 43
plot_top_k . . . 46
potato_true_ranking . . . 47
potato_visual . . . 48
potato_weighing . . . 48
predict_top_k . . . 49
print.BayesMallows . . . 50
sample_mallows . . . 51
sample_prior . . . 53
set_compute_options . . . 54
set_initial_values . . . 56
set_model_options . . . 57
set_priors . . . 58
set_progress_report . . . 59
set_smc_options . . . 60
setup_rank_data . . . 62
sounds . . . 64
sushi_rankings . . . 65
update_mallows . . . 66

Index **70**

`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, ...)

## S3 method for class 'BayesMallows'
assess_convergence(
  model_fit,
  parameter = c("alpha", "rho", "Rtilde", "cluster_probs", "theta"),
  items = NULL,
  assessors = NULL,
  ...
)

## S3 method for class 'BayesMallowsMixtures'
```

```
assess_convergence(
  model_fit,
  parameter = c("alpha", "cluster_probs"),
  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().
...	Other arguments passed on to other methods. Currently not used.
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$data$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".

Examples

```
set.seed(1)
# Fit a model on the potato_visual data
mod <- compute_mallows(setup_rank_data(potato_visual))
# Check for convergence
assess_convergence(mod)
assess_convergence(mod, parameter = "rho", items = 1:20)
```

---

assign_cluster *Assign Assessors to Clusters*

Description

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

Usage

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

Arguments

model_fit	An object of type BayesMallows, returned from compute_mallows().

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

Other posterior quantities: compute_consensus(), compute_posterior_intervals(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot.SMCMallows(), plot_elbow(), plot_top_k(), predict_top_k(), print.BayesMallows()

Examples

```
# Fit a model with three clusters to the simulated example data
set.seed(1)
mixture_model <- compute_mallows(
  data = setup_rank_data(cluster_data),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(nmc = 5000, burnin = 1000)
)

head(assign_cluster(mixture_model))
head(assign_cluster(mixture_model, soft = FALSE))
```

---

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.

See Also

Other datasets: bernoulli_data, cluster_data, potato_true_ranking, potato_visual, potato_weighing, sounds, sushi_rankings

---

bernoulli_data *Simulated intransitive pairwise preferences*

Description

Simulated dataset based on the potato_visual data. Based on the rankings in potato_visual, all n-choose-2 = 190 pairs of items were sampled from each assessor. With probability .9, the pairwise preference was in agreement with potato_visual, and with probability .1, they were in disagreement. Hence, the data generating mechanism was a Bernoulli error model (Crispino et al. 2019) with $\theta = 0.1$.

Usage

```
bernoulli_data
```

Format

An object of class `data.frame` with 2280 rows and 3 columns.

See Also

Other datasets: beach_preferences, cluster_data, potato_true_ranking, potato_visual, potato_weighing, sounds, sushi_rankings

---

burnin *See the burnin*

Description

See the current burnin value of the model.

Usage

```
burnin(model, ...)

## S3 method for class 'BayesMallows'
burnin(model, ...)

## S3 method for class 'BayesMallowsMixtures'
burnin(model, ...)

## S3 method for class 'SMCMallows'
burnin(model, ...)
```

Arguments

model	A model object.
...	Optional arguments passed on to other methods. Currently not used.

Value

An integer specifying the burnin, if it exists. Otherwise NULL.

See Also

Other modeling: burnin<-(), compute_mallows(), compute_mallows_mixtures(), compute_mallows_sequentially(), sample_prior(), update_mallows()

Examples

```
set.seed(445)
mod <- compute_mallows(setup_rank_data(potato_visual))
assess_convergence(mod)
burnin(mod)
burnin(mod) <- 1500
burnin(mod)
plot(mod)
#'
models <- compute_mallows_mixtures(
  data = setup_rank_data(cluster_data),
  n_clusters = 1:3)
burnin(models)
burnin(models) <- 100
burnin(models)
burnin(models) <- c(100, 300, 200)
burnin(models)
```

`burnin<-` *Set the burnin*

Description

Set or update the burnin of a model computed using Metropolis-Hastings.

Usage

```
burnin(model, ...) <- value

## S3 replacement method for class 'BayesMallows'
burnin(model, ...) <- value

## S3 replacement method for class 'BayesMallowsMixtures'
burnin(model, ...) <- value
```

Arguments

`model`	An object of class `BayesMallows` returned from `compute_mallows()` or an object of class `BayesMallowsMixtures` returned from `compute_mallows_mixtures()`.
`...`	Optional arguments passed on to other methods. Currently not used.
`value`	An integer specifying the burnin. If `model` is of class `BayesMallowsMixtures`, a single value will be assumed to be the burnin for each model element. Alternatively, `value` can be specified as an integer vector of the same length as `model`, and hence a separate burnin can be set for each number of mixture components.

Value

An object of class `BayesMallows` with burnin set.

See Also

Other modeling: `burnin()`, `compute_mallows()`, `compute_mallows_mixtures()`, `compute_mallows_sequentially()`, `sample_prior()`, `update_mallows()`

Examples

```
set.seed(445)
mod <- compute_mallows(setup_rank_data(potato_visual))
assess_convergence(mod)
burnin(mod)
burnin(mod) <- 1500
burnin(mod)
plot(mod)
#'
models <- compute_mallows_mixtures(
  data = setup_rank_data(cluster_data),
```

```
  n_clusters = 1:3)
burnin(models)
burnin(models) <- 100
burnin(models)
burnin(models) <- c(100, 300, 200)
burnin(models)
```

`cluster_data` *Simulated clustering data*

Description

Simulated dataset of 60 complete rankings of five items, with three different clusters.

Usage

```
cluster_data
```

Format

An object of class `matrix` (inherits from `array`) with 60 rows and 5 columns.

See Also

Other datasets: `beach_preferences`, `bernoulli_data`, `potato_true_ranking`, `potato_visual`, `potato_weighing`, `sounds`, `sushi_rankings`

`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, ...)

## S3 method for class 'BayesMallows'
compute_consensus(
  model_fit,
  type = c("CP", "MAP"),
  parameter = c("rho", "Rtilde"),
```

```
  assessors = 1L,
  ...
)

## S3 method for class 'SMCMallows'
compute_consensus(model_fit, type = c("CP", "MAP"), parameter = "rho", ...)
```

Arguments

model_fit	A model fit.
...	Other arguments passed on to other methods. Currently not used.
type	Character string specifying which consensus to compute. Either "CP" or "MAP". Defaults to "CP".
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.

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.

See Also

Other posterior quantities: assign_cluster(), compute_posterior_intervals(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot.SMCMallows(), plot_elbow(), plot_top_k(), predict_top_k(), print.BayesMallows()

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(setup_rank_data(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
burnin(model_fit) <- 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(
    setup_rank_data(sushi_rankings),
    model_options = set_model_options(n_clusters = 5))
  # Keeping the burnin at 1000, we can compute the consensus ranking per cluster
  burnin(model_fit) <- 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(setup_rank_data(preferences = beach_preferences))
  # We set burnin = 1000
  burnin(model_fit) <- 1000
  # We now compute the MAP consensus
  map_consensus_df <- compute_consensus(model_fit, type = "MAP")

  # CP CONSENSUS FOR AUGMENTED RANKINGS
  # We use the example dataset with beach preferences.
  model_fit <- compute_mallows(
    setup_rank_data(preferences = beach_preferences),
    compute_options = set_compute_options(save_aug = TRUE, aug_thinning = 2))
  # We set burnin = 1000
  burnin(model_fit) <- 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(
    setup_rank_data(preferences = beach_preferences),
    compute_options = set_compute_options(
      nmc = 1005, save_aug = TRUE, aug_thinning = 1))
  burnin(model_fit) <- 1000
  compute_consensus(model_fit, type = "MAP", parameter = "Rtilde",
                    assessors = 2L)
```

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

`compute_exact_partition_function` *Compute exact partition function*

Description

For Cayley, Hamming, and Kendall distances, computationally tractable functions are available for the exact partition function.

Usage

```
compute_exact_partition_function(
  alpha,
  n_items,
  metric = c("cayley", "hamming", "kendall")
)
```

Arguments

`alpha`	Dispersion parameter.
`n_items`	Number of items.
`metric`	Distance function, one of "cayley", "hamming", or "kendall".

Value

The logarithm of the partition function.

References

There are no references for Rd macro `\insertAllCites` on this help page.

See Also

Other partition function: `estimate_partition_function()`, `get_cardinalities()`

Examples

```
compute_exact_partition_function(
  alpha = 3.4, n_items = 34, metric = "cayley"
)

compute_exact_partition_function(
  alpha = 3.4, n_items = 34, metric = "hamming"
)

compute_exact_partition_function(
  alpha = 3.4, n_items = 34, metric = "kendall"
)
```

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

Description

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

Usage

```
compute_expected_distance(
  alpha,
  n_items,
  metric = c("footrule", "spearman", "cayley", "hamming", "kendall", "ulam")
)
```

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", "footrule", and "spearman".

Value

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

See Also

Other rank functions: compute_observation_frequency(), compute_rank_distance(), create_ranking(), get_mallows_loglik(), sample_mallows()

Examples

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

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):

$$p(r|\alpha,\rho) = \frac{1}{Z_n(\alpha)} \exp\left\{\frac{-\alpha}{n} d(r,\rho)\right\}$$

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. 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(
  data,
  model_options = set_model_options(),
  compute_options = set_compute_options(),
  priors = set_priors(),
  initial_values = set_initial_values(),
  pfun_estimate = NULL,
  progress_report = set_progress_report(),
  cl = NULL
)
```

Arguments

data	An object of class "BayesMallowsData" returned from setup_rank_data().
model_options	An object of class "BayesMallowsModelOptions" returned from set_model_options().
compute_options	An object of class "BayesMallowsComputeOptions" returned from set_compute_options().
priors	An object of class "BayesMallowsPriors" returned from set_priors().
initial_values	An object of class "BayesMallowsInitialValues" returned from set_initial_values().
pfun_estimate	Object returned from estimate_partition_function(). Defaults to NULL, and will only be used for footrule, Spearman, or Ulam distances when the cardinalities are not available, cf. get_cardinalities().

progress_report
: An object of class "BayesMallowsProgressReported" returned from `set_progress_report()`.

cl
: Optional cluster returned from `parallel::makeCluster()`. If provided, chains will be run in parallel, one on each node of `cl`.

Value

An object of class BayesMallows.

References

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

Other modeling: `burnin()`, `burnin<-()`, `compute_mallows_mixtures()`, `compute_mallows_sequentially()`, `sample_prior()`, `update_mallows()`

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:
set.seed(1)
model_fit <- compute_mallows(
  data = setup_rank_data(rankings = potato_visual),
  compute_options = set_compute_options(nmc = 2000)
  )

# We study the trace plot of the parameters
assess_convergence(model_fit, parameter = "alpha")
assess_convergence(model_fit, parameter = "rho", items = 1:4)

# Based on these plots, we set burnin = 1000.
burnin(model_fit) <- 1000
# Next, we use the generic plot function to study the posterior distributions
# of alpha and rho
plot(model_fit, parameter = "alpha")
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
```

```
compute_posterior_intervals(model_fit, parameter = "rho")

# ANALYSIS OF PAIRWISE PREFERENCES
# The example dataset beach_preferences contains pairwise
# preferences between beaches stated by 60 assessors. There
# is a total of 15 beaches in the dataset.
beach_data <- setup_rank_data(
  preferences = beach_preferences
)
# We then run the Bayesian Mallows rank model
# We save the augmented data for diagnostics purposes.
model_fit <- compute_mallows(
  data = beach_data,
  compute_options = set_compute_options(save_aug = TRUE),
  progress_report = set_progress_report(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(get_transitive_closure(beach_data), 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.


# 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
  model_fit <- compute_mallows(
    data = setup_rank_data(sushi_rankings),
    model_options = set_model_options(n_clusters = 3),
    compute_options = set_compute_options(nmc = 10000),
    progress_report = set_progress_report(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:
  burnin(model_fit) <- 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,
    data = setup_rank_data(rankings = sushi_rankings),
    compute_options = set_compute_options(
      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
  burnin(models) <- 1000
  plot_elbow(models)
  # 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
# observation_frequency argument allows providing a rankings matrix with the
# unique set of rankings, and the observation_frequency 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 observation_frequency vector
## Not run:
  set.seed(1234)
  observation_frequency <- sample.int(n = 5, size = nrow(potato_visual), replace = TRUE)
  m <- compute_mallows(
   setup_rank_data(rankings = potato_visual, observation_frequency = observation_frequency))

  # INTRANSITIVE PAIRWISE PREFERENCES
  set.seed(1234)
  mod <- compute_mallows(
    setup_rank_data(preferences = bernoulli_data),
    compute_options = set_compute_options(nmc = 5000),
```

```
  priors = set_priors(kappa = c(1, 10)),
  model_options = set_model_options(error_model = "bernoulli")
)

assess_convergence(mod)
assess_convergence(mod, parameter = "theta")
burnin(mod) <- 3000

plot(mod)
plot(mod, parameter = "theta")

## End(Not run)
# CHEKING FOR LABEL SWITCHING
## Not run:
  # This example shows how to assess if label switching happens in 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
  # NOTE: For computational efficiency, we use much fewer MCMC iterations than one
  # would normally do.
  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(
    setup_rank_data(rankings = sushi_rankings),
    model_options = set_model_options(n_clusters = 3),
    compute_options = set_compute_options(nmc = 500, save_ind_clus = FALSE)))
  # With this options, compute_mallows will save cluster_probs2.csv,
  # cluster_probs3.csv, ..., cluster_probs[nmc].csv.
  system.time(m <- compute_mallows(
    setup_rank_data(rankings = sushi_rankings),
    model_options = set_model_options(n_clusters = 3),
    compute_options = set_compute_options(nmc = 500, save_ind_clus = TRUE)))

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

  # We set the burnin to 200
  burnin <- 200

  # 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(
    regmatches(x = cluster_files,m = regexpr(pattern = "[0-9]+", cluster_files)
               ))
```

```
  # 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$data$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$value))
  z$chain <- NULL
  z <- reshape(z, direction = "wide", idvar = "iteration", timevar = "assessor")
  z$iteration <- NULL
  z <- as.matrix(z)

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

  # Check the proportion of cluster assignments that were switched
  mean(apply(switch_check$permutations$STEPHENS, 1, function(x) {
    !all(x == seq(1, m$n_clusters, by = 1))
  }))

  # 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)
```

`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,
  data,
  model_options = set_model_options(),
  compute_options = set_compute_options(),
  priors = set_priors(),
  initial_values = set_initial_values(),
  pfun_estimate = NULL,
  progress_report = set_progress_report(),
  cl = NULL
)
```

Arguments

n_clusters	Integer vector specifying the number of clusters to use.
data	An object of class "BayesMallowsData" returned from setup_rank_data().
model_options	An object of class "BayesMallowsModelOptions" returned from set_model_options().
compute_options	An object of class "BayesMallowsComputeOptions" returned from set_compute_options().
priors	An object of class "BayesMallowsPriors" returned from set_priors().
initial_values	An object of class "BayesMallowsInitialValues" returned from set_initial_values().
pfun_estimate	Object returned from estimate_partition_function(). Defaults to NULL, and will only be used for footrule, Spearman, or Ulam distances when the cardinalities are not available, cf. get_cardinalities().
progress_report	An object of class "BayesMallowsProgressReported" returned from set_progress_report().
cl	Optional cluster returned from parallel::makeCluster(). If provided, chains will be run in parallel, one on each node of cl.

Details

The n_clusters argument to set_model_options() is ignored when calling compute_mallows_mixtures.

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().

See Also

Other modeling: burnin(), burnin<-(), compute_mallows(), compute_mallows_sequentially(), sample_prior(), update_mallows()

Examples

```
# SIMULATED CLUSTER DATA
set.seed(1)
n_clusters <- seq(from = 1, to = 5)
models <- compute_mallows_mixtures(
  n_clusters = n_clusters, data = setup_rank_data(cluster_data),
  compute_options = set_compute_options(nmc = 2000, include_wcd = TRUE))

# There is good convergence for 1, 2, and 3 cluster, but not for 5.
# Also note that there seems to be label switching around the 7000th iteration
# for the 2-cluster solution.
assess_convergence(models)
# We can create an elbow plot, suggesting that there are three clusters, exactly
# as simulated.
burnin(models) <- 1000
plot_elbow(models)

# We now fit a model with three clusters
mixture_model <- compute_mallows(
  data = setup_rank_data(cluster_data),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(nmc = 2000))

# The trace plot for this model looks good. It seems to converge quickly.
assess_convergence(mixture_model)
# We set the burnin to 500
burnin(mixture_model) <- 500

# We can now look at posterior quantities
# Posterior of scale parameter alpha
plot(mixture_model)
plot(mixture_model, parameter = "rho", items = 4:5)
# There is around 33 % probability of being in each cluster, in agreemeent
# with the data simulating mechanism
plot(mixture_model, parameter = "cluster_probs")
# We can also look at a cluster assignment plot
plot(mixture_model, parameter = "cluster_assignment")

# 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, data = setup_rank_data(sushi_rankings),
    compute_options = set_compute_options(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
  burnin(models) <- 1000
  plot_elbow(models)
  # 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
  mixture_model <- compute_mallows(
    rankings = sushi_rankings,
    model_options = set_model_options(n_clusters = 5),
    compute_options = set_compute_options(include_wcd = TRUE))
  # Delete the models object to free some memory
  rm(models)
  # Set the burnin
  burnin(mixture_model) <- 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,
    compute_options = set_compute_options(include_wcd = TRUE),
    cl = cl)
  stopCluster(cl)

## End(Not run)
```

compute_mallows_sequentially *Estimate the Bayesian Mallows Model Sequentially*

Description

Compute the posterior distributions of the parameters of the Bayesian Mallows model using sequential Monte Carlo. This is based on the algorithms developed in Stein (2023). This function differs from `update_mallows()` in that it takes all the data at once, and uses SMC to fit the model step-by-step. Used in this way, SMC is an alternative to Metropolis-Hastings, which may work better in some settings. In addition, it allows visualization of the learning process.

Usage

```
compute_mallows_sequentially(
  data,
  initial_values,
  model_options = set_model_options(),
  smc_options = set_smc_options(),
  compute_options = set_compute_options(),
  priors = set_priors(),
  pfun_estimate = NULL
)
```

Arguments

`data`	A list of objects of class "BayesMallowsData" returned from `setup_rank_data()`. Each list element is interpreted as the data belonging to a given timepoint.
`initial_values`	An object of class "BayesMallowsPriorSamples" returned from `sample_prior()`.
`model_options`	An object of class "BayesMallowsModelOptions" returned from `set_model_options()`.
`smc_options`	An object of class "SMCOptions" returned from `set_smc_options()`.
`compute_options`	An object of class "BayesMallowsComputeOptions" returned from `set_compute_options()`.
`priors`	An object of class "BayesMallowsPriors" returned from `set_priors()`.
`pfun_estimate`	Object returned from `estimate_partition_function()`. Defaults to `NULL`, and will only be used for footrule, Spearman, or Ulam distances when the cardinalities are not available, cf. `get_cardinalities()`.

Details

This function is very new, and plotting functions and other tools for visualizing the posterior distribution do not yet work. See the examples for some workarounds.

Value

An object of class BayesMallowsSequential.

References

Stein A (2023). *Sequential Inference with the Mallows Model*. Ph.D. thesis, Lancaster University.

See Also

Other modeling: `burnin()`, `burnin<-()`, `compute_mallows()`, `compute_mallows_mixtures()`, `sample_prior()`, `update_mallows()`

Examples

```
## Not run:
# Observe one ranking at each of 12 timepoints
library(ggplot2)
data <- lapply(seq_len(nrow(potato_visual)), function(i) {
  setup_rank_data(potato_visual[i, ], user_ids = i)
})

initial_values <- sample_prior(
  n = 200, n_items = 20,
  priors = set_priors(gamma = 3, lambda = .1))

mod <- compute_mallows_sequentially(
  data = data,
  initial_values = initial_values,
  smc_options = set_smc_options(n_particles = 500, mcmc_steps = 20))

# We can see the acceptance ratio of the move step for each timepoint:
get_acceptance_ratios(mod)

plot_dat <- data.frame(
  n_obs = seq_along(data),
  alpha_mean = apply(mod$alpha_samples, 2, mean),
  alpha_sd = apply(mod$alpha_samples, 2, sd)
)

# Visualize how the dispersion parameter is being learned as more data arrive
ggplot(plot_dat, aes(x = n_obs, y = alpha_mean, ymin = alpha_mean - alpha_sd,
                     ymax = alpha_mean + alpha_sd)) +
  geom_line() +
  geom_ribbon(alpha = .1) +
  ylab(expression(alpha)) +
  xlab("Observations") +
  theme_classic() +
  scale_x_continuous(
    breaks = seq(min(plot_dat$n_obs), max(plot_dat$n_obs), by = 1))

# Visualize the learning of the rank for a given item (item 1 in this example)
plot_dat <- data.frame(
  n_obs = seq_along(data),
  rank_mean = apply(mod$rho_samples[1, , ], 2, mean),
  rank_sd = apply(mod$rho_samples[1, , ], 2, sd)
)
```

```
ggplot(plot_dat, aes(x = n_obs, y = rank_mean, ymin = rank_mean - rank_sd,
                     ymax = rank_mean + rank_sd)) +
  geom_line() +
  geom_ribbon(alpha = .1) +
  xlab("Observations") +
  ylab(expression(rho[1])) +
  theme_classic() +
  scale_x_continuous(
    breaks = seq(min(plot_dat$n_obs), max(plot_dat$n_obs), by = 1))

## End(Not run)
```

`compute_observation_frequency` *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 `observation_frequency` argument to `compute_mallows()`.

Usage

```
compute_observation_frequency(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.

See Also

Other rank functions: `compute_expected_distance()`, `compute_rank_distance()`, `create_ranking()`, `get_mallows_loglik()`, `sample_mallows()`

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
compute_observation_frequency(rankings = data0)
```

```
# The function also works when the data have missing values
rankings <- matrix(c(1, 2, 3, 4,
                     1, 2, 4, NA,
                     1, 2, 4, NA,
                     3, 2, 1, 4,
                     NA, NA, 2, 1,
                     NA, NA, 2, 1,
                     NA, NA, 2, 1,
                     2, NA, 1, NA), ncol = 4, byrow = TRUE)

compute_observation_frequency(rankings)
```

`compute_posterior_intervals` *Compute Posterior Intervals*

Description

Compute posterior intervals of parameters of interest.

Usage

```
compute_posterior_intervals(model_fit, ...)

## S3 method for class 'BayesMallows'
compute_posterior_intervals(
  model_fit,
  parameter = c("alpha", "rho", "cluster_probs"),
  level = 0.95,
  decimals = 3L,
  ...
)

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

Arguments

`model_fit`	A model object.
`...`	Other arguments. Currently not used.

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.

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.

References

There are no references for Rd macro \insertAllCites on this help page.

See Also

Other posterior quantities: assign_cluster(), compute_consensus(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot.SMCMallows(), plot_elbow(), plot_top_k(), predict_top_k(), print.BayesMallows()

Examples

```
set.seed(1)
model_fit <- compute_mallows(
  setup_rank_data(potato_visual),
  compute_options = set_compute_options(nmc = 3000, 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
  model_fit <- compute_mallows(
    setup_rank_data(sushi_rankings),
    model_options = set_model_options(n_clusters = 5))
  burnin(model_fit) <- 1000

  compute_posterior_intervals(model_fit, parameter = "alpha")

  compute_posterior_intervals(model_fit, parameter = "cluster_probs")

## End(Not run)
```

`compute_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

```
compute_rank_distance(
  rankings,
  rho,
  metric = c("footrule", "spearman", "cayley", "hamming", "kendall", "ulam"),
  observation_frequency = 1
)
```

Arguments

`rankings`	A matrix of size $N \times n_{items}$ 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"`.
`observation_frequency`	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.

See Also

Other rank functions: `compute_expected_distance()`, `compute_observation_frequency()`, `create_ranking()`, `get_mallows_loglik()`, `sample_mallows()`

Examples

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

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

---

create_ranking *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.

See Also

Other rank functions: `compute_expected_distance()`, `compute_observation_frequency()`, `compute_rank_distance()`, `get_mallows_loglik()`, `sample_mallows()`

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)
```

---

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). Note that exact partition functions can be computed efficiently for Cayley, Hamming and Kendall distances with any number of items, for footrule distances with up to 50 items, Spearman distance with up to 20 items, and Ulam distance with up to 60 items. This function is thus intended for the complement of these cases. See `get_cardinalities()` and `compute_exact_partition_function()` for details.

Usage

```
estimate_partition_function(
  method = c("importance_sampling", "asymptotic"),
  alpha_vector,
  n_items,
  metric,
  n_iterations,
  K = 20,
  cl = 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".
n_iterations	Integer specifying the number of iterations to use. When method = "importance_sampling", this is the number of Monte Carlo samples to generate. When method = "asymptotic", on the other hand, it represents the number of iterations of the IPFP algorithm.
K	Integer specifying the parameter K in the asymptotic approximation of the partition function. Only used when method = "asymptotic". Defaults to 20.
cl	Optional computing cluster used for parallelization, returned from parallel::makeCluster(). Defaults to NULL. Only used when method = "importance_sampling".

Value

A matrix with two column and number of rows equal the degree of the fitted polynomial approximating the partition function. The matrix can be supplied to the pfun_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.

See Also

Other partition function: compute_exact_partition_function(), get_cardinalities()

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"

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

```
fit1 <- estimate_partition_function(
  method = "importance_sampling", alpha_vector = alpha_vector,
  n_items = n_items, metric = metric, n_iterations = 1e3)
# A matrix containing powers of alpha and regression coefficients is returned
fit1
# The approximated partition function can hence be obtained:
estimate1 <-
  vapply(alpha_vector, function(a) sum(a^fit1[, 1] * fit1[, 2]), numeric(1))

# Now let us recompute with 2e3 Monte Carlo samples
fit2 <- estimate_partition_function(
  method = "importance_sampling", alpha_vector = alpha_vector,
  n_items = n_items, metric = metric, n_iterations = 2e3)
estimate2 <-
  vapply(alpha_vector, function(a) sum(a^fit2[, 1] * fit2[, 2]), numeric(1))

# ASYMPTOTIC APPROXIMATION
# We can also compute an estimate using the asymptotic approximation
fit3 <- estimate_partition_function(
  method = "asymptotic", alpha_vector = alpha_vector,
  n_items = n_items, metric = metric, n_iterations = 50)
estimate3 <-
  vapply(alpha_vector, function(a) sum(a^fit3[, 1] * fit3[, 2]), numeric(1))

# We can now plot the estimates side-by-side
plot(alpha_vector, estimate1, type = "l", xlab = expression(alpha),
     ylab = expression(log(Z(alpha))))
lines(alpha_vector, estimate2, col = 2)
lines(alpha_vector, estimate3, col = 3)
legend(x = 7, y = 40, legend = c("IS,1e3", "IS,2e3", "IPFP"),
       col = 1:3, lty = 1)

# 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
# pfun_estimate argument to compute_mallows
# Say we choose to use the importance sampling estimate with 1e4 Monte Carlo samples:
model_fit <- compute_mallows(
  setup_rank_data(potato_visual),
  model_options = set_model_options(metric = "spearman"),
  compute_options = set_compute_options(nmc = 200),
  pfun_estimate = fit2)


## End(Not run)
```

---

get_acceptance_ratios *Get Acceptance Ratios*

Description

Extract acceptance ratio from Metropolis-Hastings algorithm used by `compute_mallows()` or by the move step in `update_mallows()` and `compute_mallows_sequentially()`. If burnin is not set in the call to `compute_mallows()`, the acceptance ratio for all iterations will be reported. Otherwise the post burnin acceptance ratio is reported. For the SMC method the acceptance ratios apply to all iterations, since no burnin is needed in here.

Usage

```
get_acceptance_ratios(model_fit, ...)

## S3 method for class 'BayesMallows'
get_acceptance_ratios(model_fit, ...)

## S3 method for class 'SMCMallows'
get_acceptance_ratios(model_fit, ...)
```

Arguments

`model_fit`	A model fit.
`...`	Other arguments passed on to other methods. Currently not used.

Value

A list with elements `alpha_acceptance`, `rho_acceptance`, and `aug_acceptance`. Each element contains acceptance ratios (between 0 and 1) for the corresponding parameter proposals in the Metropolis-Hastings algorithm. For models with multiple chains, each element is a list with one acceptance ratio per chain. Higher values indicate higher acceptance rates for the Metropolis-Hastings proposals.

See Also

Other posterior quantities: `assign_cluster()`, `compute_consensus()`, `compute_posterior_intervals()`, `heat_plot()`, `plot.BayesMallows()`, `plot.SMCMallows()`, `plot_elbow()`, `plot_top_k()`, `predict_top_k()`, `print.BayesMallows()`

Examples

```
set.seed(1)
mod <- compute_mallows(
  data = setup_rank_data(potato_visual),
  compute_options = set_compute_options(burnin = 200)
)

get_acceptance_ratios(mod)
```

`get_cardinalities` *Get cardinalities for each distance*

Description

The partition function for the Mallows model can be defined in a computationally efficient manner as

$$Z_n(\alpha) = \sum_{d_n \in \mathcal{D}_n} N_{m,n} e^{-(\alpha/n) d_m}.$$

In this equation, $\mathcal{D}_n$ a set containing all possible distances at the given number of items, and d_m is on element of this set. Finally, $N_{m,n}$ is the number of possible configurations of the items that give the particular distance. See Irurozki et al. (2016), Vitelli et al. (2018), and Crispino et al. (2023) for details.

For footrule distance, the cardinalities come from entry A062869 in the On-Line Encyclopedia of Integer Sequences (OEIS) (Sloane and Inc. 2020). For Spearman distance, they come from entry A175929, and for Ulam distance from entry A126065.

Usage

```
get_cardinalities(n_items, metric = c("footrule", "spearman", "ulam"))
```

Arguments

`n_items`	Number of items.
`metric`	Distance function, one of "footrule", "spearman", or "ulam".

Value

A dataframe with two columns, `distance` which contains each distance in the support set at the current number of items, i.e., d_m, and `value` which contains the number of values at this particular distances, i.e., $N_{m,n}$.

References

Crispino M, Mollica C, Astuti V, Tardella L (2023). "Efficient and accurate inference for mixtures of Mallows models with Spearman distance." *Statistics and Computing*, **33**(5). ISSN 1573-1375, doi:10.1007/s11222023102668, http://dx.doi.org/10.1007/s11222-023-10266-8.

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.

Sloane NJA, Inc. TOF (2020). "The on-line encyclopedia of integer sequences." https://oeis.org/.

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

Other partition function: compute_exact_partition_function(), estimate_partition_function()

Examples

```
# Extract the cardinalities for four items with footrule distance
n_items <- 4
dat <- get_cardinalities(n_items)
# Compute the partition function at alpha = 2
alpha <- 2
sum(dat$value * exp(-alpha / n_items * dat$distance))
#'
# We can confirm that it is correct by enumerating all possible combinations
all <- expand.grid(1:4, 1:4, 1:4, 1:4)
perms <- all[apply(all, 1, function(x) length(unique(x)) == 4), ]
sum(apply(perms, 1, function(x) exp(-alpha / n_items * sum(abs(x - 1:4)))))

# We do the same for the Spearman distance
dat <- get_cardinalities(n_items, metric = "spearman")
sum(dat$value * exp(-alpha / n_items * dat$distance))
#'
# We can confirm that it is correct by enumerating all possible combinations
sum(apply(perms, 1, function(x) exp(-alpha / n_items * sum((x - 1:4)^2))))
```

---

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 = c("footrule", "spearman", "cayley", "hamming", "kendall", "ulam"),
  rankings,
  observation_frequency = NULL,
  log = TRUE
)
```

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", "footrule", and "spearman".
rankings	A matrix with observed rankings in each row.
observation_frequency	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, observation_frequency 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.

See Also

Other rank functions: compute_expected_distance(), compute_observation_frequency(), compute_rank_distance(), create_ranking(), sample_mallows()

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 <- compute_observation_frequency(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],
  observation_frequency = 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],
  observation_frequency = freq_distr[, n_items + 1],
  log = TRUE
  )
```

`get_transitive_closure` *Get transitive closure*

Description

A simple method for showing any transitive closure computed by `setup_rank_data()`.

Usage

```
get_transitive_closure(rank_data)
```

Arguments

`rank_data`	An object of class `"BayesMallowsData"` returned from setup_rank_data.

Value

A dataframe with transitive closure, if there is any.

See Also

Other preprocessing: `set_compute_options()`, `set_initial_values()`, `set_model_options()`, `set_priors()`, `set_progress_report()`, `set_smc_options()`, `setup_rank_data()`

Examples

```
# Original beach preferences
head(beach_preferences)
dim(beach_preferences)
# We then create a rank data object
dat <- setup_rank_data(preferences = beach_preferences)
# The transitive closure contains additional filled-in preferences implied
# by the stated preferences.
head(get_transitive_closure(dat))
dim(get_transitive_closure(dat))
```

---

heat_plot *Heat plot of posterior probabilities*

Description

Generates a heat plot with items in their consensus ordering along the horizontal axis and ranking along the vertical axis. The color denotes posterior probability.

Usage

```
heat_plot(model_fit, ...)
```

Arguments

model_fit	An object of type BayesMallows, returned from compute_mallows().
...	Additional arguments passed on to other methods. In particular, type = "CP" or type = "MAP" can be passed on to compute_consensus() to determine the order of items along the horizontal axis.

Details

In models with a single cluster, the items are sorted along the x-axis according to the consensus ranking. In models with more than one cluster, the items are sorted alphabetically.

Value

A ggplot object.

See Also

Other posterior quantities: assign_cluster(), compute_consensus(), compute_posterior_intervals(), get_acceptance_ratios(), plot.BayesMallows(), plot.SMCMallows(), plot_elbow(), plot_top_k(), predict_top_k(), print.BayesMallows()

Examples

```
set.seed(1)
model_fit <- compute_mallows(
  setup_rank_data(potato_visual),
  compute_options = set_compute_options(nmc = 2000, burnin = 500))

heat_plot(model_fit)
heat_plot(model_fit, type = "MAP")

## Model with three clusters
mod <- compute_mallows(
  data = setup_rank_data(rankings = cluster_data),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(nmc = 10000, burnin = 1000)
)

heat_plot(mod)
heat_plot(mod, type = "MAP")
```

`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, parameter = "alpha", items = NULL, ...)
```

Arguments

`x`	An object of type `BayesMallows`, returned from `compute_mallows()`.
`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$data$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).

See Also

Other posterior quantities: `assign_cluster()`, `compute_consensus()`, `compute_posterior_intervals()`, `get_acceptance_ratios()`, `heat_plot()`, `plot.SMCMallows()`, `plot_elbow()`, `plot_top_k()`, `predict_top_k()`, `print.BayesMallows()`

Examples

```
model_fit <- compute_mallows(setup_rank_data(potato_visual))
burnin(model_fit) <- 1000

# By default, the scale parameter "alpha" is plotted
plot(model_fit)
# 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"))

## Not run:
  # Plots of mixture parameters:
  model_fit <- compute_mallows(
    setup_rank_data(sushi_rankings),
    model_options = set_model_options(n_clusters = 5))
  burnin(model_fit) <- 1000
  # Posterior distributions of the cluster probabilities
  plot(model_fit, parameter = "cluster_probs")
  # Cluster assignment plot. Color shows the probability of belonging to each
  # cluster.
  plot(model_fit, parameter = "cluster_assignment")

## End(Not run)
```

`plot.SMCMallows` *Plot SMC Posterior Distributions*

Description

Plot posterior distributions of SMC-Mallow parameters.

Usage

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

Arguments

x	An object of type SMC-Mallows.
parameter	Character string defining the parameter to plot. Available options are "alpha" and "rho".
items	Either a vector of item names, or a vector of indices. If NULL, five items are selected randomly.
...	Other arguments passed to plot (not used).

Value

A plot of the posterior distributions

See Also

Other posterior quantities: assign_cluster(), compute_consensus(), compute_posterior_intervals(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot_elbow(), plot_top_k(), predict_top_k(), print.BayesMallows()

Examples

```
## Not run:
set.seed(1)
# UPDATING A MALLOWS MODEL WITH NEW COMPLETE RANKINGS
# Assume we first only observe the first four rankings in the potato_visual
# dataset
data_first_batch <- potato_visual[1:4, ]

# We start by fitting a model using Metropolis-Hastings
mod_init <- compute_mallows(
  data = setup_rank_data(data_first_batch),
  compute_options = set_compute_options(nmc = 10000))

# Convergence seems good after no more than 2000 iterations
assess_convergence(mod_init)
burnin(mod_init) <- 2000

# Next, assume we receive four more observations
data_second_batch <- potato_visual[5:8, ]

# We can now update the model using sequential Monte Carlo
mod_second <- update_mallows(
  model = mod_init,
  new_data = setup_rank_data(rankings = data_second_batch),
  smc_options = set_smc_options(resampler = "systematic")
  )

# This model now has a collection of particles approximating the posterior
# distribution after the first and second batch
# We can use all the posterior summary functions as we do for the model
# based on compute_mallows():
```

```
plot(mod_second)
plot(mod_second, parameter = "rho", items = 1:4)
compute_posterior_intervals(mod_second)

# Next, assume we receive the third and final batch of data. We can update
# the model again
data_third_batch <- potato_visual[9:12, ]
mod_final <- update_mallows(
  model = mod_second, new_data = setup_rank_data(rankings = data_third_batch))

# We can plot the same things as before
plot(mod_final)
compute_consensus(mod_final)

# UPDATING A MALLOWS MODEL WITH NEW OR UPDATED PARTIAL RANKINGS
# The sequential Monte Carlo algorithm works for data with missing ranks as
# well. This both includes the case where new users arrive with partial ranks,
# and when previously seen users arrive with more complete data than they had
# previously.
# We illustrate for top-k rankings of the first 10 users in potato_visual
potato_top_10 <- ifelse(potato_visual[1:10, ] > 10, NA_real_,
                        potato_visual[1:10, ])
potato_top_12 <- ifelse(potato_visual[1:10, ] > 12, NA_real_,
                        potato_visual[1:10, ])
potato_top_14 <- ifelse(potato_visual[1:10, ] > 14, NA_real_,
                        potato_visual[1:10, ])

# We need the rownames as user IDs
(user_ids <- 1:10)

# First, users provide top-10 rankings
mod_init <- compute_mallows(
  data = setup_rank_data(rankings = potato_top_10, user_ids = user_ids),
  compute_options = set_compute_options(nmc = 10000))

# Convergence seems fine. We set the burnin to 2000.
assess_convergence(mod_init)
burnin(mod_init) <- 2000

# Next assume the users update their rankings, so we have top-12 instead.
mod1 <- update_mallows(
  model = mod_init,
  new_data = setup_rank_data(rankings = potato_top_12, user_ids = user_ids),
  smc_options = set_smc_options(resampler = "stratified")
)

plot(mod1)

# Then, assume we get even more data, this time top-14 rankings:
mod2 <- update_mallows(
  model = mod1,
  new_data = setup_rank_data(rankings = potato_top_14, user_ids = user_ids)
)
```

```
plot(mod2)

# Finally, assume a set of new users arrive, who have complete rankings.
potato_new <- potato_visual[11:12, ]
# We need to update the user IDs, to show that these users are different
(user_ids <- 11:12)

mod_final <- update_mallows(
  model = mod2,
  new_data = setup_rank_data(rankings = potato_new, user_ids = user_ids)
)

plot(mod_final)

# We can also update models with pairwise preferences
# We here start by running MCMC on the first 20 assessors of the beach data
# A realistic application should run a larger number of iterations than we
# do in this example.
set.seed(3)
dat <- subset(beach_preferences, assessor <= 20)
mod <- compute_mallows(
  data = setup_rank_data(
    preferences = beach_preferences),
  compute_options = set_compute_options(nmc = 3000, burnin = 1000)
)

# Next we provide assessors 21 to 24 one at a time. Note that we sample the
# initial augmented rankings in each particle for each assessor from 200
# different topological sorts consistent with their transitive closure.
for(i in 21:24){
  mod <- update_mallows(
    model = mod,
    new_data = setup_rank_data(
      preferences = subset(beach_preferences, assessor == i),
      user_ids = i),
    smc_options = set_smc_options(latent_sampling_lag = 0,
                                  max_topological_sorts = 200)
  )
}

# Compared to running full MCMC, there is a downward bias in the scale
# parameter. This can be alleviated by increasing the number of particles,
# MCMC steps, and the latent sampling lag.
plot(mod)
compute_consensus(mod)

## End(Not run)
```

---

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(...)
```

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(). Alternatively an object returned from compute_mallows_mixtures().

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

Other posterior quantities: assign_cluster(), compute_consensus(), compute_posterior_intervals(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot.SMCMallows(), plot_top_k(), predict_top_k(), print.BayesMallows()

Examples

```
# SIMULATED CLUSTER DATA
set.seed(1)
n_clusters <- seq(from = 1, to = 5)
models <- compute_mallows_mixtures(
  n_clusters = n_clusters, data = setup_rank_data(cluster_data),
  compute_options = set_compute_options(nmc = 2000, include_wcd = TRUE))

# There is good convergence for 1, 2, and 3 cluster, but not for 5.
# Also note that there seems to be label switching around the 7000th iteration
# for the 2-cluster solution.
assess_convergence(models)
# We can create an elbow plot, suggesting that there are three clusters, exactly
# as simulated.
burnin(models) <- 1000
plot_elbow(models)

# We now fit a model with three clusters
mixture_model <- compute_mallows(
  data = setup_rank_data(cluster_data),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(nmc = 2000))

# The trace plot for this model looks good. It seems to converge quickly.
```

```
assess_convergence(mixture_model)
# We set the burnin to 500
burnin(mixture_model) <- 500

# We can now look at posterior quantities
# Posterior of scale parameter alpha
plot(mixture_model)
plot(mixture_model, parameter = "rho", items = 4:5)
# There is around 33 % probability of being in each cluster, in agreemeent
# with the data simulating mechanism
plot(mixture_model, parameter = "cluster_probs")
# We can also look at a cluster assignment plot
plot(mixture_model, parameter = "cluster_assignment")

# 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, data = setup_rank_data(sushi_rankings),
    compute_options = set_compute_options(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
  burnin(models) <- 1000
  plot_elbow(models)
  # 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
  mixture_model <- compute_mallows(
    rankings = sushi_rankings,
    model_options = set_model_options(n_clusters = 5),
    compute_options = set_compute_options(include_wcd = TRUE))
  # Delete the models object to free some memory
  rm(models)
  # Set the burnin
  burnin(mixture_model) <- 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,
    compute_options = set_compute_options(include_wcd = TRUE),
    cl = cl)
  stopCluster(cl)

## End(Not run)
```

`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, k = 3)
```

Arguments

`model_fit`	An object of type `BayesMallows`, returned from `compute_mallows()`.
`k`	Integer specifying the k in top-k.

See Also

Other posterior quantities: `assign_cluster()`, `compute_consensus()`, `compute_posterior_intervals()`, `get_acceptance_ratios()`, `heat_plot()`, `plot.BayesMallows()`, `plot.SMCMallows()`, `plot_elbow()`, `predict_top_k()`, `print.BayesMallows()`

Examples

```
set.seed(1)
# 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(
  data = setup_rank_data(preferences = beach_preferences),
  compute_options = set_compute_options(
    nmc = 1000, burnin = 500, save_aug = TRUE))
# By default, the probability of being top-3 is plotted
# The default plot gives the probability for each assessor
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)

# We can also plot for clusters
model_fit <- compute_mallows(
  data = setup_rank_data(preferences = beach_preferences),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(
    nmc = 1000, burnin = 500, save_aug = TRUE)
  )
# The modal ranking in general differs between clusters, but the plot still
# represents the posterior distribution of each user's augmented rankings.
plot_top_k(model_fit)
```

---

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.

See Also

Other datasets: beach_preferences, bernoulli_data, cluster_data, potato_visual, potato_weighing, sounds, sushi_rankings

---

potato_visual *Potato weights assessed visually*

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.

See Also

Other datasets: beach_preferences, bernoulli_data, cluster_data, potato_true_ranking, potato_weighing, sounds, sushi_rankings

---

potato_weighing *Potato weights assessed by hand*

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.

See Also

Other datasets: beach_preferences, bernoulli_data, cluster_data, potato_true_ranking, potato_visual, sounds, sushi_rankings

---

`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, k = 3)
```

Arguments

model_fit	An object of type BayesMallows, returned from compute_mallows().
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

Other posterior quantities: assign_cluster(), compute_consensus(), compute_posterior_intervals(), get_acceptance_ratios(), heat_plot(), plot.BayesMallows(), plot.SMCMallows(), plot_elbow(), plot_top_k(), print.BayesMallows()

Examples

```
set.seed(1)
# 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(
  data = setup_rank_data(preferences = beach_preferences),
  compute_options = set_compute_options(
    nmc = 1000, burnin = 500, save_aug = TRUE))
```

```
# By default, the probability of being top-3 is plotted
# The default plot gives the probability for each assessor
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)

# We can also plot for clusters
model_fit <- compute_mallows(
  data = setup_rank_data(preferences = beach_preferences),
  model_options = set_model_options(n_clusters = 3),
  compute_options = set_compute_options(
    nmc = 1000, burnin = 500, save_aug = TRUE)
  )
# The modal ranking in general differs between clusters, but the plot still
# represents the posterior distribution of each user's augmented rankings.
plot_top_k(model_fit)
```

`print.BayesMallows` *Print Method for BayesMallows Objects*

Description

The default print method for a `BayesMallows` object.

Usage

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

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

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

Arguments

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

See Also

Other posterior quantities: `assign_cluster()`, `compute_consensus()`, `compute_posterior_intervals()`, `get_acceptance_ratios()`, `heat_plot()`, `plot.BayesMallows()`, `plot.SMCMallows()`, `plot_elbow()`, `plot_top_k()`, `predict_top_k()`

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.

See Also

Other rank functions: `compute_expected_distance()`, `compute_observation_frequency()`, `compute_rank_distance()`, `create_ranking()`, `get_mallows_loglik()`

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(
```

```
  setup_rank_data(samples),
  compute_options = set_compute_options(nmc = 10000))
# The highest posterior density interval covers alpha0 = 10.
burnin(model_fit) <- 2000
compute_posterior_intervals(model_fit, parameter = "alpha")
```

`sample_prior` *Sample from prior distribution*

Description

Function to obtain samples from the prior distributions of the Bayesian Mallows model. Intended to be given to `update_mallows()`.

Usage

```
sample_prior(n, n_items, priors = set_priors())
```

Arguments

`n`	An integer specifying the number of samples to take.
`n_items`	An integer specifying the number of items to be ranked.
`priors`	An object of class "BayesMallowsPriors" returned from `set_priors()`.

Value

An object of class "BayesMallowsPriorSample", containing `n` independent samples of α and ρ.

See Also

Other modeling: `burnin()`, `burnin<-()`, `compute_mallows()`, `compute_mallows_mixtures()`, `compute_mallows_sequentially()`, `update_mallows()`

Examples

```
## Not run:
# We can use a collection of particles from the prior distribution as
# initial values for the sequential Monte Carlo algorithm.
# Here we start by drawing 1000 particles from the priors, using default
# parameters.
prior_samples <- sample_prior(1000, ncol(sushi_rankings))
# Next, we provide the prior samples to update_mallws(), together
# with the first five rows of the sushi dataset
model1 <- update_mallows(
  model = prior_samples,
  new_data = setup_rank_data(sushi_rankings[1:5, ]))
plot(model1)
```

```
# We keep adding more data
model2 <- update_mallows(
  model = model1,
  new_data = setup_rank_data(sushi_rankings[6:10, ]))
plot(model2)

model3 <- update_mallows(
  model = model2,
  new_data = setup_rank_data(sushi_rankings[11:15, ]))
plot(model3)

## End(Not run)
```

`set_compute_options` *Specify options for computation*

Description

Set parameters related to the Metropolis-Hastings algorithm.

Usage

```
set_compute_options(
  nmc = 2000,
  burnin = NULL,
  alpha_prop_sd = 0.1,
  rho_proposal = c("ls", "swap"),
  leap_size = 1,
  aug_method = c("uniform", "pseudo"),
  pseudo_aug_metric = c("footrule", "spearman"),
  swap_leap = 1,
  alpha_jump = 1,
  aug_thinning = 1,
  clus_thinning = 1,
  rho_thinning = 1,
  include_wcd = FALSE,
  save_aug = FALSE,
  save_ind_clus = FALSE
)
```

Arguments

`nmc`	Integer specifying the number of iteration of the Metropolis-Hastings algorithm to run. Defaults to `2000`. See `assess_convergence()` for tools to check convergence of the Markov chain.
`burnin`	Integer defining the number of samples to discard. Defaults to `NULL`, which means that burn-in is not set.

`alpha_prop_sd` Numeric value specifying the σ parameter of the lognormal proposal distribution used for α in the Metropolis-Hastings algorithm. The logarithm of the proposed samples will have standard deviation given by `alpha_prop_sd`. Defaults to `0.1`.

`rho_proposal` Character string specifying the proposal distribution of modal ranking ρ. Defaults to "ls", which means that the leap-and-shift algorithm of Vitelli et al. (2018) is used. The other option is "swap", which means that the swap proposal of Crispino et al. (2019) is used instead.

`leap_size` Integer specifying the step size of the distribution defined in `rho_proposal` for proposing new latent ranks rho. Defaults to 1.

`aug_method` Augmentation proposal for use with missing data. One of "pseudo" and "uniform". Defaults to "uniform", which means that new augmented rankings are proposed by sampling uniformly from the set of available ranks, see Section 4 in Vitelli et al. (2018). Setting the argument to "pseudo" instead, means that the pseudo-likelihood proposal defined in Chapter 5 of Stein (2023) is used instead.

`pseudo_aug_metric`
String defining the metric to be used in the pseudo-likelihood proposal. Only used if `aug_method = "pseudo"`. Can be either "footrule" or "spearman", and defaults to "footrule".

`swap_leap` Integer specifying the leap size for the swap proposal used for proposing latent ranks in the case of non-transitive pairwise preference data. Note that leap size for the swap proposal when used for proposal the modal ranking ρ is given by the `leap_size` argument above.

`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 1.

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

`clus_thinning` Integer specifying the thinning to be applied to cluster assignments and cluster probabilities. Defaults to 1.

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

`include_wcd` Logical indicating whether to store the within-cluster distances computed during the Metropolis-Hastings algorithm. Defaults to `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_thinning`th 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()`.

`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.

Value

An object of class "BayesMallowsComputeOptions", to be provided in the compute_options argument to compute_mallows(), compute_mallows_mixtures(), or update_mallows().

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.

Stein A (2023). *Sequential Inference with the Mallows Model*. Ph.D. thesis, Lancaster University.

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

Other preprocessing: get_transitive_closure(), set_initial_values(), set_model_options(), set_priors(), set_progress_report(), set_smc_options(), setup_rank_data()

---

set_initial_values *Set initial values of scale parameter and modal ranking*

Description

Set initial values used by the Metropolis-Hastings algorithm.

Usage

```
set_initial_values(rho_init = NULL, alpha_init = 1)
```

Arguments

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.
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. When chains are run in parallel, by providing an argument cl = cl, then alpha_init can be a vector of of length length(cl), each element of which becomes an initial value for the given chain.

Value

An object of class "BayesMallowsInitialValues", to be provided to the initial_values argument of compute_mallows() or compute_mallows_mixtures().

See Also

Other preprocessing: get_transitive_closure(), set_compute_options(), set_model_options(), set_priors(), set_progress_report(), set_smc_options(), setup_rank_data()

---

set_model_options *Set options for Bayesian Mallows model*

Description

Specify various model options for the Bayesian Mallows model.

Usage

```
set_model_options(
  metric = c("footrule", "spearman", "cayley", "hamming", "kendall", "ulam"),
  n_clusters = 1,
  error_model = c("none", "bernoulli")
)
```

Arguments

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.
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.
error_model	Character string specifying which model to use for inconsistent rankings. Defaults to "none", 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.

Value

An object of class "BayesMallowsModelOptions", to be provided in the model_options argument to compute_mallows(), compute_mallows_mixtures(), or update_mallows().

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.

See Also

Other preprocessing: `get_transitive_closure()`, `set_compute_options()`, `set_initial_values()`, `set_priors()`, `set_progress_report()`, `set_smc_options()`, `setup_rank_data()`

---

set_priors *Set prior parameters for Bayesian Mallows model*

Description

Set values related to the prior distributions for the Bayesian Mallows model.

Usage

```
set_priors(gamma = 1, lambda = 0.001, psi = 10, kappa = c(1, 3))
```

Arguments

`gamma`	Strictly positive numeric value specifying the shape parameter of the gamma prior distribution of α. Defaults to 1, thus recovering the exponential prior distribution used by (Vitelli et al. 2018).
`lambda`	Strictly positive numeric value specifying the rate parameter of the gamma prior distribution of α. Defaults to `0.001`. When `n_cluster > 1`, each mixture component α_c has the same prior distribution.
`psi`	Positive 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.
`kappa`	Hyperparameters of the truncated Beta prior used for error probability θ in the Bernoulli error model. The prior has the form $\pi(\theta) = \theta^{\kappa_1}(1-\theta)^{\kappa_2}$. Defaults to `c(1, 3)`, which means that the θ is a priori expected to be closer to zero than to 0.5. See (Crispino et al. 2019) for details.

Value

An object of class `"BayesMallowsPriors"`, to be provided in the `priors` argument to `compute_mallows()`, `compute_mallows_mixtures()`, or `update_mallows()`.

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.

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

Other preprocessing: `get_transitive_closure()`, `set_compute_options()`, `set_initial_values()`, `set_model_options()`, `set_progress_report()`, `set_smc_options()`, `setup_rank_data()`

`set_progress_report` *Set progress report options for MCMC algorithm*

Description

Specify whether progress should be reported, and how often.

Usage

```
set_progress_report(verbose = FALSE, report_interval = 1000)
```

Arguments

`verbose`	Boolean specifying whether to report progress or not. Defaults to `FALSE`.
`report_interval`	Strictly positive number specifying how many iterations of MCMC should be run between each progress report. Defaults to `1000`.

Value

An object of class `"BayesMallowsProgressReport"`, to be provided in the `progress_report` argument to `compute_mallows()` and `compute_mallows_mixtures()`.

References

There are no references for Rd macro `\insertAllCites` on this help page.

See Also

Other preprocessing: `get_transitive_closure()`, `set_compute_options()`, `set_initial_values()`, `set_model_options()`, `set_priors()`, `set_smc_options()`, `setup_rank_data()`

`set_smc_options` *Set SMC compute options*

Description

Sets the SMC compute options to be used in `update_mallows.BayesMallows()`.

Usage

```
set_smc_options(
  n_particles = 1000,
  mcmc_steps = 5,
  resampler = c("stratified", "systematic", "residual", "multinomial"),
  latent_sampling_lag = NA_integer_,
  max_topological_sorts = 1
)
```

Arguments

`n_particles`	Integer specifying the number of particles.
`mcmc_steps`	Number of MCMC steps to be applied in the resample-move step.
`resampler`	Character string defining the resampling method to use. One of "stratified", "systematic", "residual", and "multinomial". Defaults to "stratified". While multinomial resampling was used in Stein (2023), stratified, systematic, or residual resampling typically give lower Monte Carlo error (Douc and Cappe 2005; Hol et al. 2006; Naesseth et al. 2019).
`latent_sampling_lag`	Parameter specifying the number of timesteps to go back when resampling the latent ranks in the move step. See Section 6.2.3 of (Kantas et al. 2015) for details. The L in their notation corresponds to `latent_sampling_lag`. See more under Details. Defaults to `NA`, which means that all latent ranks from previous timesteps are moved. If set to `0`, no move step is applied to the latent ranks.
`max_topological_sorts`	User when pairwise preference data are provided, and specifies the maximum number of topological sorts of the graph corresponding to the transitive closure for each user will be used as initial ranks. Defaults to 1, which means that all particles get the same initial augmented ranking. If larger than 1, the initial augmented ranking for each particle will be sampled from a set of maximum size `max_topological_sorts`. If the actual number of topological sorts consists of fewer rankings, then this determines the upper limit.

Value

An object of class "SMCOptions".

Lag parameter in move step

The parameter `latent_sampling_lag` corresponds to L in (Kantas et al. 2015). Its use in this package is can be explained in terms of Algorithm 12 in (Stein 2023). The relevant line of the algorithm is:

for $j = 1 : M_t$ **do**
M-H step: update $\tilde{\mathbf{R}}_j^{(i)}$ with proposal $\tilde{\mathbf{R}}_j' \sim q(\tilde{\mathbf{R}}_j^{(i)} | \mathbf{R}_j, \boldsymbol{\rho}_t^{(i)}, \alpha_t^{(i)})$.
end

Let L denote the value of `latent_sampling_lag`. With this parameter, we modify for algorithm so it becomes

for $j = M_{t-L+1} : M_t$ **do**
M-H step: update $\tilde{\mathbf{R}}_j^{(i)}$ with proposal $\tilde{\mathbf{R}}_j' \sim q(\tilde{\mathbf{R}}_j^{(i)} | \mathbf{R}_j, \boldsymbol{\rho}_t^{(i)}, \alpha_t^{(i)})$.
end

This means that with $L = 0$ no move step is performed on any latent ranks, whereas $L = 1$ means that the move step is only applied to the parameters entering at the given timestep. The default, `latent_sampling_lag = NA` means that $L = t$ at each timestep, and hence all latent ranks are part of the move step at each timestep.

References

Douc R, Cappe O (2005). "Comparison of resampling schemes for particle filtering." In *ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005.*. doi:10.1109/ispa.2005.195385, http://dx.doi.org/10.1109/ISPA.2005.195385.

Hol JD, Schon TB, Gustafsson F (2006). "On Resampling Algorithms for Particle Filters." In *2006 IEEE Nonlinear Statistical Signal Processing Workshop*. doi:10.1109/nsspw.2006.4378824, http://dx.doi.org/10.1109/NSSPW.2006.4378824.

Kantas N, Doucet A, Singh SS, Maciejowski J, Chopin N (2015). "On Particle Methods for Parameter Estimation in State-Space Models." *Statistical Science*, **30**(3). ISSN 0883-4237, doi:10.1214/14sts511, http://dx.doi.org/10.1214/14-STS511.

Naesseth CA, Lindsten F, Schön TB (2019). "Elements of Sequential Monte Carlo." *Foundations and Trends® in Machine Learning*, **12**(3), 187–306. ISSN 1935-8245, doi:10.1561/2200000074, http://dx.doi.org/10.1561/2200000074.

Stein A (2023). *Sequential Inference with the Mallows Model*. Ph.D. thesis, Lancaster University.

See Also

Other preprocessing: `get_transitive_closure()`, `set_compute_options()`, `set_initial_values()`, `set_model_options()`, `set_priors()`, `set_progress_report()`, `setup_rank_data()`

`setup_rank_data` *Setup rank data*

Description

Prepare rank or preference data for further analyses.

Usage

```
setup_rank_data(
  rankings = NULL,
  preferences = NULL,
  user_ids = numeric(),
  observation_frequency = NULL,
  validate_rankings = TRUE,
  na_action = c("augment", "fail", "omit"),
  cl = NULL,
  max_topological_sorts = 1,
  timepoint = NULL,
  n_items = 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. 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 to `set_model_options()`. A vector length `n_items` is silently converted to a matrix of length `1 x n_items`, and names (if any), are used as column names.

`preferences` 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.
- `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

`user_ids` Optional `numeric` vector of user IDs. Only only used by `update_mallows()`. If provided, new data can consist of updated partial rankings from users already in the dataset, as described in Section 6 of Stein (2023).

`observation_frequency`
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, `observation_frequency` must have the same number of elements as there are rows in `rankings`, and `rankings` cannot be `NULL`. See `compute_observation_frequency()` for a convenience function for computing it.

`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 `NA`s in `rankings`, and `"omit"` which simply deletes rows with `NA`s in them.

`cl` Optional computing cluster used for parallelization when generating transitive closure based on preferences, returned from `parallel::makeCluster()`. Defaults to `NULL`.

`max_topological_sorts`
When preference data are provided, multiple rankings will be consistent with the preferences stated by each users. These rankings are the topological sorts of the directed acyclic graph corresponding to the transitive closure of the preferences. This number defaults to one, which means that the algorithm stops when it finds a single initial ranking which is compatible with the rankings stated by the user. By increasing this number, multiple rankings compatible with the pairwise preferences will be generated, and one initial value will be sampled from this set.

`timepoint` Integer vector specifying the timepoint. Defaults to `NULL`, which means that a vector of ones, one for each observation, is generated. Used by `update_mallows()` to identify data with a given iteration of the sequential Monte Carlo algorithm. If not `NULL`, must contain one integer for each row in `rankings`.

`n_items` Integer specifying the number of items. Defaults to `NULL`, which means that the number of items is inferred from `rankings` or from `preferences`. Setting `n_items` manually can be useful with pairwise preference data in the SMC algorithm, i.e., when `rankings` is `NULL` and `preferences` is non-`NULL`, and contains a small number of pairwise preferences for a subset of users and items.

Value

An object of class "BayesMallowsData", to be provided in the data argument to compute_mallows().

Note

Setting max_topological_sorts larger than 1 means that many 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 mix properly.

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

References

Stein A (2023). *Sequential Inference with the Mallows Model*. Ph.D. thesis, Lancaster University.

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

Other preprocessing: get_transitive_closure(), set_compute_options(), set_initial_values(), set_model_options(), set_priors(), set_progress_report(), set_smc_options()

---

sounds *Sounds data*

Description

Data from an experiment in which 46 individuals compared 12 different sounds (Barrett and Crispino 2018). Each assessor was asked multiple times to compare a pair of two sounds, indicating which of the sounds sounded the most like it was human generated. The pairwise preference for each assessor are in general non-transitive. These data inspired the Mallows model for non-transitive pairwise preferences developed by (Crispino et al. 2019).

Usage

```
sounds
```

Format

An object of class `data.frame` with 1380 rows and 3 columns.

References

Barrett N, Crispino M (2018). "The impact of 3-D sound spatialisation on listeners' understanding of human agency in acousmatic music." *Journal of New Music Research*, **47**(5), 399–415. doi:10.1080/09298215.2018.1437187.

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.

See Also

Other datasets: `beach_preferences`, `bernoulli_data`, `cluster_data`, `potato_true_ranking`, `potato_visual`, `potato_weighing`, `sushi_rankings`

`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.

See Also

Other datasets: `beach_preferences`, `bernoulli_data`, `cluster_data`, `potato_true_ranking`, `potato_visual`, `potato_weighing`, `sounds`

update_mallows *Update a Bayesian Mallows model with new users*

Description

Update a Bayesian Mallows model estimated using the Metropolis-Hastings algorithm in `compute_mallows()` using the sequential Monte Carlo algorithm described in Stein (2023).

Usage

```
update_mallows(model, new_data, ...)

## S3 method for class 'BayesMallowsPriorSamples'
update_mallows(
  model,
  new_data,
  model_options = set_model_options(),
  smc_options = set_smc_options(),
  compute_options = set_compute_options(),
  priors = model$priors,
  pfun_estimate = NULL,
  ...
)

## S3 method for class 'BayesMallows'
update_mallows(
  model,
  new_data,
  model_options = set_model_options(),
  smc_options = set_smc_options(),
  compute_options = set_compute_options(),
  priors = model$priors,
  ...
)

## S3 method for class 'SMCMallows'
update_mallows(model, new_data, ...)
```

Arguments

`model`	A model object of class "BayesMallows" returned from `compute_mallows()`, an object of class "SMCMallows" returned from this function, or an object of class "BayesMallowsPriorSamples" returned from `sample_prior()`.
`new_data`	An object of class "BayesMallowsData" returned from `setup_rank_data()`. The object should contain the new data being provided.
`...`	Optional arguments. Currently not used.

model_options	An object of class "BayesMallowsModelOptions" returned from set_model_options().
smc_options	An object of class "SMCOptions" returned from set_smc_options().
compute_options	An object of class "BayesMallowsComputeOptions" returned from set_compute_options().
priors	An object of class "BayesMallowsPriors" returned from set_priors(). Defaults to the priors used in model.
pfun_estimate	Object returned from estimate_partition_function(). Defaults to NULL, and will only be used for footrule, Spearman, or Ulam distances when the cardinalities are not available, cf. get_cardinalities(). Only used by the specialization for objects of type "BayesMallowsPriorSamples".

Value

An updated model, of class "SMCMallows".

See Also

Other modeling: burnin(), burnin<-(), compute_mallows(), compute_mallows_mixtures(), compute_mallows_sequentially(), sample_prior()

Examples

```
## Not run:
set.seed(1)
# UPDATING A MALLOWS MODEL WITH NEW COMPLETE RANKINGS
# Assume we first only observe the first four rankings in the potato_visual
# dataset
data_first_batch <- potato_visual[1:4, ]

# We start by fitting a model using Metropolis-Hastings
mod_init <- compute_mallows(
  data = setup_rank_data(data_first_batch),
  compute_options = set_compute_options(nmc = 10000))

# Convergence seems good after no more than 2000 iterations
assess_convergence(mod_init)
burnin(mod_init) <- 2000

# Next, assume we receive four more observations
data_second_batch <- potato_visual[5:8, ]

# We can now update the model using sequential Monte Carlo
mod_second <- update_mallows(
  model = mod_init,
  new_data = setup_rank_data(rankings = data_second_batch),
  smc_options = set_smc_options(resampler = "systematic")
  )

# This model now has a collection of particles approximating the posterior
# distribution after the first and second batch
```

```
# We can use all the posterior summary functions as we do for the model
# based on compute_mallows():
plot(mod_second)
plot(mod_second, parameter = "rho", items = 1:4)
compute_posterior_intervals(mod_second)

# Next, assume we receive the third and final batch of data. We can update
# the model again
data_third_batch <- potato_visual[9:12, ]
mod_final <- update_mallows(
  model = mod_second, new_data = setup_rank_data(rankings = data_third_batch))

# We can plot the same things as before
plot(mod_final)
compute_consensus(mod_final)

# UPDATING A MALLOWS MODEL WITH NEW OR UPDATED PARTIAL RANKINGS
# The sequential Monte Carlo algorithm works for data with missing ranks as
# well. This both includes the case where new users arrive with partial ranks,
# and when previously seen users arrive with more complete data than they had
# previously.
# We illustrate for top-k rankings of the first 10 users in potato_visual
potato_top_10 <- ifelse(potato_visual[1:10, ] > 10, NA_real_,
                        potato_visual[1:10, ])
potato_top_12 <- ifelse(potato_visual[1:10, ] > 12, NA_real_,
                        potato_visual[1:10, ])
potato_top_14 <- ifelse(potato_visual[1:10, ] > 14, NA_real_,
                        potato_visual[1:10, ])

# We need the rownames as user IDs
(user_ids <- 1:10)

# First, users provide top-10 rankings
mod_init <- compute_mallows(
  data = setup_rank_data(rankings = potato_top_10, user_ids = user_ids),
  compute_options = set_compute_options(nmc = 10000))

# Convergence seems fine. We set the burnin to 2000.
assess_convergence(mod_init)
burnin(mod_init) <- 2000

# Next assume the users update their rankings, so we have top-12 instead.
mod1 <- update_mallows(
  model = mod_init,
  new_data = setup_rank_data(rankings = potato_top_12, user_ids = user_ids),
  smc_options = set_smc_options(resampler = "stratified")
)

plot(mod1)

# Then, assume we get even more data, this time top-14 rankings:
mod2 <- update_mallows(
  model = mod1,
```

```
  new_data = setup_rank_data(rankings = potato_top_14, user_ids = user_ids)
)

plot(mod2)

# Finally, assume a set of new users arrive, who have complete rankings.
potato_new <- potato_visual[11:12, ]
# We need to update the user IDs, to show that these users are different
(user_ids <- 11:12)

mod_final <- update_mallows(
  model = mod2,
  new_data = setup_rank_data(rankings = potato_new, user_ids = user_ids)
)

plot(mod_final)

# We can also update models with pairwise preferences
# We here start by running MCMC on the first 20 assessors of the beach data
# A realistic application should run a larger number of iterations than we
# do in this example.
set.seed(3)
dat <- subset(beach_preferences, assessor <= 20)
mod <- compute_mallows(
  data = setup_rank_data(
    preferences = beach_preferences),
  compute_options = set_compute_options(nmc = 3000, burnin = 1000)
)

# Next we provide assessors 21 to 24 one at a time. Note that we sample the
# initial augmented rankings in each particle for each assessor from 200
# different topological sorts consistent with their transitive closure.
for(i in 21:24){
  mod <- update_mallows(
    model = mod,
    new_data = setup_rank_data(
      preferences = subset(beach_preferences, assessor == i),
      user_ids = i),
    smc_options = set_smc_options(latent_sampling_lag = 0,
                                  max_topological_sorts = 200)
  )
}

# Compared to running full MCMC, there is a downward bias in the scale
# parameter. This can be alleviated by increasing the number of particles,
# MCMC steps, and the latent sampling lag.
plot(mod)
compute_consensus(mod)

## End(Not run)
```

Index

∗ **datasets**
beach_preferences, 5
bernoulli_data, 6
cluster_data, 9
potato_true_ranking, 47
potato_visual, 48
potato_weighing, 48
sounds, 64
sushi_rankings, 65
∗ **diagnostics**
assess_convergence, 3
∗ **modeling**
burnin, 6
burnin<-, 8
compute_mallows, 14
compute_mallows_mixtures, 19
compute_mallows_sequentially, 23
sample_prior, 53
update_mallows, 66
∗ **partition function**
compute_exact_partition_function, 12
estimate_partition_function, 30
get_cardinalities, 34
∗ **posterior quantities**
assign_cluster, 4
compute_consensus, 9
compute_posterior_intervals, 26
get_acceptance_ratios, 32
heat_plot, 38
plot.BayesMallows, 39
plot.SMCMallows, 40
plot_elbow, 43
plot_top_k, 46
predict_top_k, 49
print.BayesMallows, 50
∗ **preprocessing**
get_transitive_closure, 37
set_compute_options, 54
set_initial_values, 56
set_model_options, 57
set_priors, 58
set_progress_report, 59
set_smc_options, 60
setup_rank_data, 62
∗ **rank functions**
compute_expected_distance, 13
compute_observation_frequency, 25
compute_rank_distance, 28
create_ranking, 29
get_mallows_loglik, 35
sample_mallows, 51

assess_convergence, 3
assess_convergence(), *54*
assign_cluster, 4, *10*, *27*, *33*, *38, 39*, *41*, *44*, *46*, *49, 50*

beach_preferences, 5, *6*, *9*, *48, 49*, *65*
bernoulli_data, 6, *6*, *9*, *48, 49*, *65*
burnin, 6, *8*, *15*, *20*, *24*, *53*, *67*
burnin<-, 8

cluster_data, *6*, 9, *48, 49*, *65*
compute_consensus, *5*, 9, *27*, *33*, *38, 39*, *41*, *44*, *46*, *49, 50*
compute_consensus(), *38*
compute_exact_partition_function, 12, *31*, *35*
compute_exact_partition_function(), *30*
compute_expected_distance, 13, *25*, *28, 29*, *36*, *52*
compute_mallows, *7, 8*, 14, *20*, *24*, *53*, *67*
compute_mallows(), *4*, *8*, *25*, *31*, *33*, *38, 39*, *44*, *46*, *49, 50*, *56–59*, *64*, *66*
compute_mallows_mixtures, *7, 8*, *15*, 19, *24*, *53*, *67*
compute_mallows_mixtures(), *4*, *8*, *44*, *56–59*

compute_mallows_sequentially, *7, 8*, *15*, *20*, 23, *53*, *67*
compute_mallows_sequentially(), *33*
compute_observation_frequency, *13*, 25, *28, 29*, *36*, *52*
compute_observation_frequency(), *63*
compute_posterior_intervals, *5*, *10*, 26, *33*, *38, 39*, *41*, *44*, *46*, *49, 50*
compute_rank_distance, *13*, *25*, 28, *29*, *36*, *52*
create_ordering *(*create_ranking*)*, 29
create_ranking, *13*, *25*, *28*, 29, *36*, *52*
create_ranking(), *62*

estimate_partition_function, *12*, 30, *35*
estimate_partition_function(), *14*, *20*, *23*, *67*

get_acceptance_ratios, *5*, *10*, *27*, 32, *38, 39*, *41*, *44*, *46*, *49, 50*
get_cardinalities, *12*, *31*, 34
get_cardinalities(), *14*, *20*, *23*, *30*, *67*
get_mallows_loglik, *13*, *25*, *28, 29*, 35, *52*
get_transitive_closure, 37, *56–59*, *61*, *64*

heat_plot, *5*, *10*, *27*, *33*, 38, *39*, *41*, *44*, *46*, *49, 50*

parallel::makeCluster(), *15*, *20*, *31*, *63*
plot, *41*
plot.BayesMallows, *5*, *10*, *27*, *33*, *38*, 39, *41*, *44*, *46*, *49, 50*
plot.SMCMallows, *5*, *10*, *27*, *33*, *38, 39*, 40, *44*, *46*, *49, 50*
plot_elbow, *5*, *10*, *27*, *33*, *38, 39*, *41*, 43, *46*, *49, 50*
plot_elbow(), *20*, *55*
plot_top_k, *5*, *10*, *27*, *33*, *38, 39*, *41*, *44*, 46, *49, 50*
plot_top_k(), *55*
potato_true_ranking, *6*, *9*, 47, *48, 49*, *65*
potato_visual, *6*, *9*, 48, *48, 49*, *65*
potato_weighing, *6*, *9*, 48, *48*, *65*
predict_top_k, *5*, *10*, *27*, *33*, *38, 39*, *41*, *44*, *46*, 49, *50*
print.BayesMallows, *5*, *10*, *27*, *33*, *38, 39*, *41*, *44*, *46*, *49*, 50
print.BayesMallowsMixtures *(*print.BayesMallows*)*, 50
print.SMCMallows *(*print.BayesMallows*)*, 50

sample_mallows, *13*, *25*, *28, 29*, *36*, 51
sample_prior, *7, 8*, *15*, *20*, *24*, 53, *67*
sample_prior(), *23*, *66*
set_compute_options, *37*, 54, *57–59*, *61*, *64*
set_compute_options(), *14*, *20*, *23*, *67*
set_initial_values, *37*, 56, *56*, *58, 59*, *61*, *64*
set_initial_values(), *14*, *20*
set_model_options, *37*, *56*, 57, *57*, *59*, *61*, *64*
set_model_options(), *14*, *20*, *23*, *62*, *67*
set_priors, *37*, *56, 57*, 58, *58, 59*, *61*, *64*
set_priors(), *14*, *20*, *23*, *53*, *67*
set_progress_report, *37*, *56–58*, 59, *59*, *61*, *64*
set_progress_report(), *15*, *20*
set_smc_options, *37*, *56–59*, 60, *64*
set_smc_options(), *23*, *67*
setup_rank_data, *37*, *56–59*, *61*, 62
setup_rank_data(), *14*, *20*, *23*, *37*, *66*
sounds, *6*, *9*, *48, 49*, 64, *65*
sushi_rankings, *6*, *9*, *48, 49*, 65, *65*

update_mallows, *7, 8*, *15*, *20*, *24*, *53*, 66
update_mallows(), *23*, *33*, *53*, *56–58*, *63*
update_mallows.BayesMallows(), *60*